虚拟内存管理（页表）

概述

目标

（1）掌握虚拟内存管理相关的概念
（2）了解多级页表的管理方法
（3）掌握异常处理与内核模块联动
（4）实现用户内存的懒惰分配特性
（5）实现写时拷贝的进程创建

简述

简述：虚拟内存管理是计算机操作系统中的一种技术，用于管理和映射程序的逻辑地址空间到物理内存。它通过使用页表来实现逻辑地址到物理地址的转换，并提供了以下重要功能：地址空间扩展、内存保护、内存共享以及内存管理和分配。

虚拟内存管理将程序的逻辑地址空间划分为固定大小的页，并将物理内存划分为相同大小的页框。通过页表，虚拟内存管理系统可以将逻辑页与物理页框之间建立映射关系。当程序访问逻辑地址时，虚拟内存管理系统根据页表将其转换为对应的物理地址。

虚拟内存管理的主要优势之一是可以将程序的地址空间扩展到比物理内存更大的范围，使得程序可以运行更大规模的任务。此外，通过设置页表项的权限位，可以实现内存的保护，防止未授权的访问。虚拟内存管理还支持内存共享，允许多个进程将相同的物理页映射到各自的地址空间，实现共享内存的机制。同时，虚拟内存管理系统可以根据需求将页从磁盘交换到物理内存或者将页从物理内存换出到磁盘，以实现内存的有效管理和分配。

总之，虚拟内存管理通过使用页表实现了逻辑地址到物理地址的转换，并提供了诸多功能，包括地址空间扩展、内存保护、内存共享和内存管理等，为程序提供了一个抽象的、连续的地址空间，提高了内存利用效率和系统的可靠性。

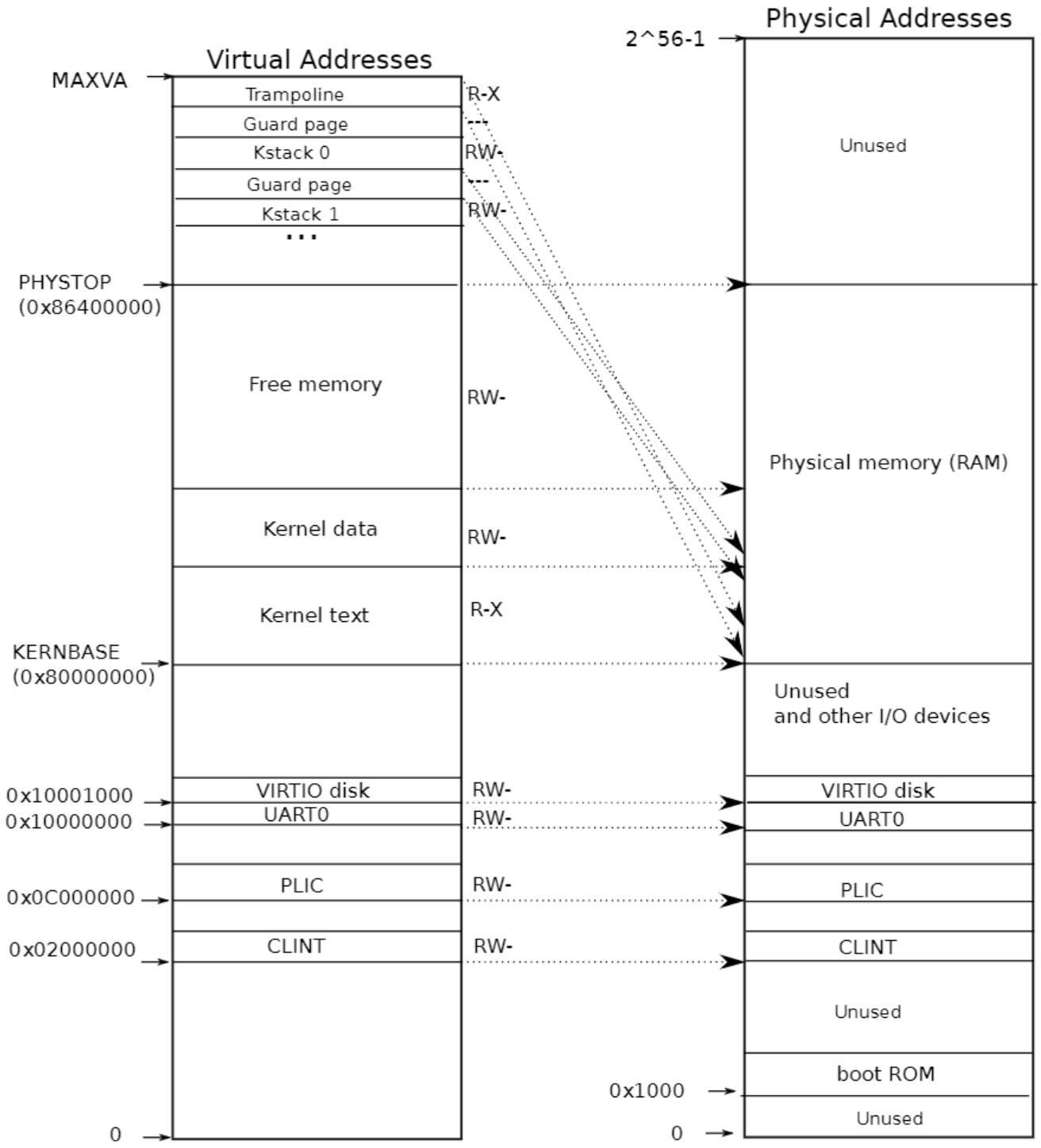
讲解

为了使各个进程相互隔离，互不影响，操作系统要使用一种机制，使得每个进程认为好像自己在独占整个内存，这种机制就是页表（Page Table）。

这就是 OSTEP 里提到的虚拟化的体现。

Address Spaces

内核的地址空间如下图。内核使用“直接映射”获取内存和内存映射设备寄存器；也就是说，将资源映射到等于物理地址的虚拟地址。



有几个内核虚拟地址不是直接映射：

- 蹦床页面 (trampoline page)。它映射在虚拟地址空间的顶部；用户页表具有相同的映射。第 4 章讨论了蹦床页面的作用，但我们在这里看到了一个有趣的页表用例；一个物理页面（持有蹦床代码）在内核的虚拟地址空间中映射了两次：一次在虚拟地址空间的顶部，一次直接映射。
- 内核栈页面。每个进程都有自己的内核栈，它将映射到偏高一些的地址，这样 xv6 在它之下就可以留下一个未映射的保护页 (guard page)。保护页的 PTE 是无效的（也就是说 `PTE_V` 没有设置），所以如果内核溢出内核栈就会引发一个异常，内核触发 `panic`。如果没有保护页，栈溢出将会覆盖其他内核内存，引发错误操作。恐慌崩溃（panic crash）是更可取的方案。 *（注：Guard page 不会浪费物理内存，它只是占据了虚拟地址空间的一段靠后的地址，但并不映射到物理地址空间。）*

用户进程的地址空间如下图，进程的地址空间是连续的，但实际映射到内存里，并不一定是连续的（xv6 中分配以 page[4Kb]为最小单位，page 内是连续的，page 间不确定）。用户进程在内存中的实际位置在上图中的 `Free memory` 部分。

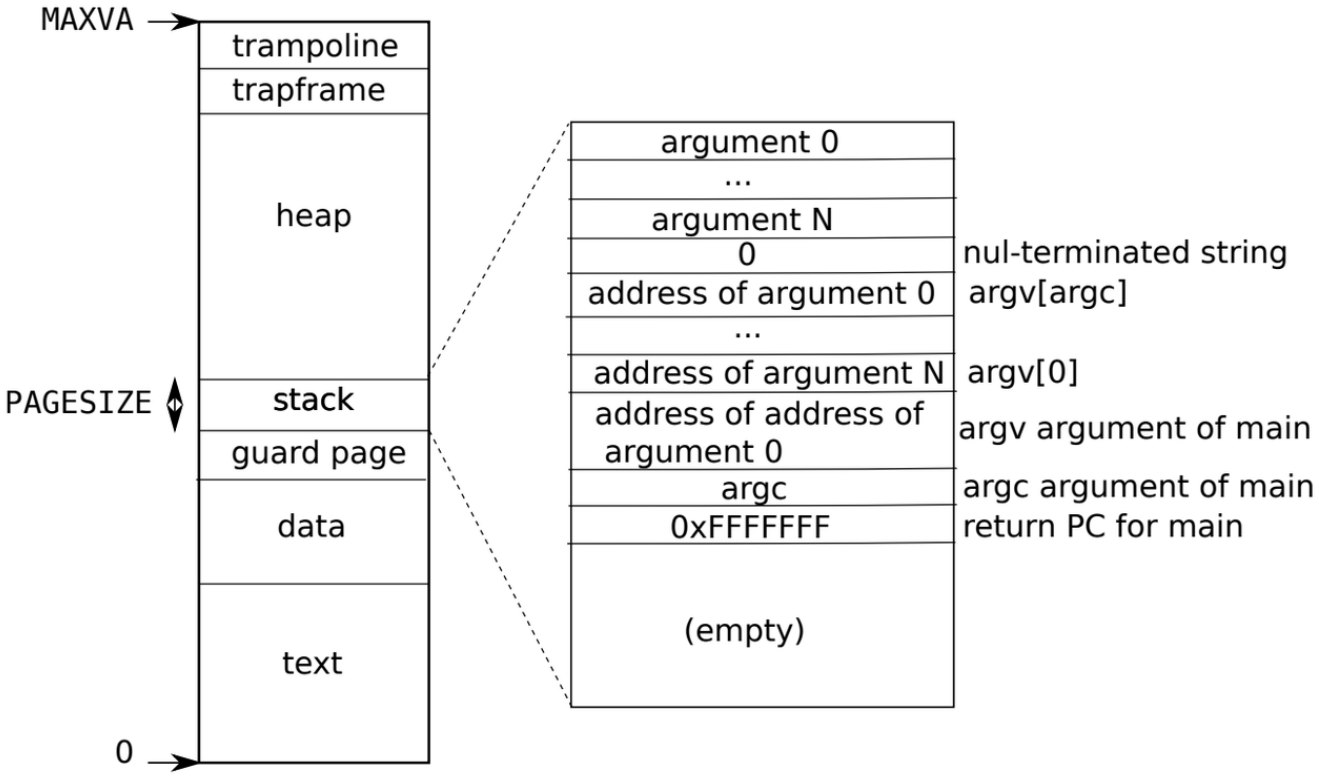


Figure 3.4: A process's user address space, with its initial stack.

- text：also known as a code segment。存放编译后二进制代码。
- data：全局变量以及 static 变量
- guard page：用来防止栈溢出。这个段不会被映射到物理地址，对这个段进行读写会导致异常。
- stack：固定大小的用户栈空间，向下增长
- heap：expandable heap，malloc 就在管理这片内存
- trapframe：用来保存必要的寄存器，以在 user/kernel mode 之间进行切换，每个进程都有单独的一个。
- trampoline：处理 trap 的代码位置

Page Table

我们如何能够实现地址空间呢？或者说如何在一个物理内存上，创建不同的地址空间？

最常见的方法，同时也是非常灵活的一种方法就是使用页表（Page Tables）。

页表是在硬件中通过处理器和 MMU（Memory Management Unit）实现。MMU 是硬件的一部分而不是操作系统的一部分。

假设 CPU 正在执行指令，例如 `sd $7, (a0)` 。对于任何一条带有地址的指令，其中的地址应该认为是虚拟内存地址而不是物理地址。假设寄存器 a0 中是地址 0x1000，那么这是一个虚拟内存地址。

虚拟内存地址会被转到内存管理单元（MMU，Memory Management Unit）。内存管理单元会将虚拟地址翻译成物理地址。

从 CPU 的角度来说，一旦 MMU 打开了，它执行的每条指令中的地址都是虚拟内存地址。

为了能够完成虚拟内存地址到物理内存地址的翻译，MMU 会有一个表单，表单中，一边是虚拟内存地址，另一边是物理内存地址。

通常来说，内存地址对应关系的表单也保存在内存中。所以 CPU 中需要有一些寄存器用来存放表单在物理内存中的地址。现在，在内存的某个位置保存了地址关系表单，我们假设这个位置的物理内存地址是 0x10。那么在 RISC-V 上一个叫做**SATP**的寄存器会保存地址 0x10。

MMU 并不会保存 page table，page table 保存在内存中，MMU 只是会去查看 page table 完成翻译。

每个进程都有内存中有他自己的 page table，这个分配是由操作系统完成的，因此在进程切换时**SATP**寄存器也会被更改。

RISC-V Page Table

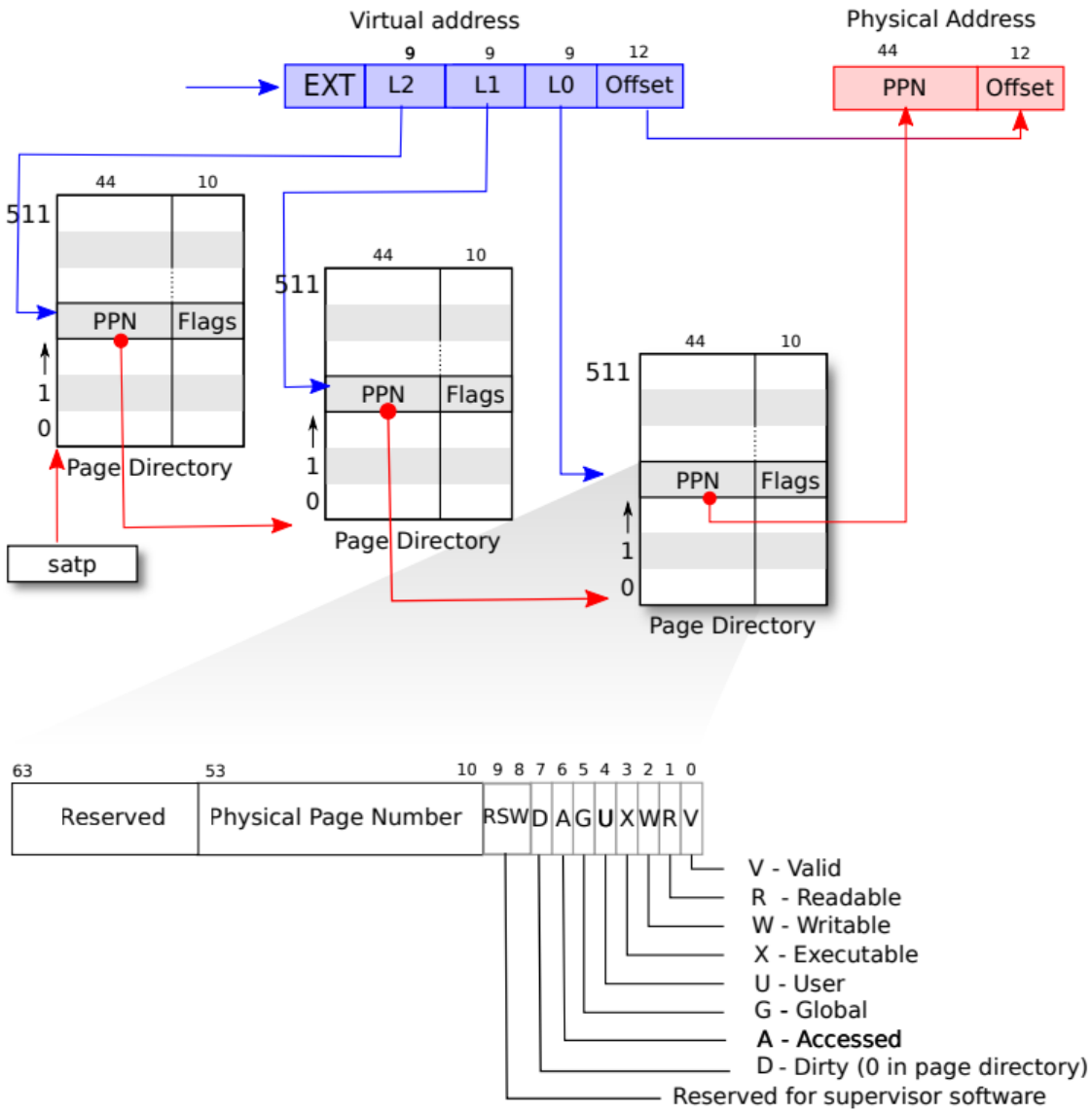


Figure 3.2: RISC-V address translation details.

- V：PTE 是否有效，当为 0 时 PTE 无效，剩余的其他 bits 无意义，可以被随意使用。
- RWX：readable, writable, executable
  - 当三个 bits 都是 0 时，该 PTE 指向下一层 page table。否则，该 PTE 为叶节点。
- U：在 User mode 下是否可以访问该 page。U=1，U-mode 下可以访问。
  - 此外，该标志位也对 S-mode 存在限制。
  - 当 `sstatus` 寄存器中的 `SUM (permit Supervisor User Memory access)` 字段为 1 时，S-mode 下可以读写 U = 1 的 page。否则不行。（U=0 的 page，S-mode 随意访问）。
  - 无论如何，S-mode 下不能执行 U = 1 page 中的代码。
- G：全局映射标志
  - 对于非叶 PTE，全局设置意味着页表后续级别中的所有映射都是全局的。
  - 请注意，未能将全局映射标记为全局映射只会降低性能，而将非全局映射标记为全局映射是一个软件错误。
- A：The A bit indicates the virtual page has been read, written, or fetched from since the last time the A bit was cleared.
- D：The D bit indicates the virtual page has been written since the last time the D bit was cleared.
- For **non-leaf** PTEs, the D, A, and U bits are reserved for future standard use and **must be cleared** by software for forward compatibility.

XV6 使用 Sv39 RISC-V，也就是 64 位地址中前 25 位不使用 (必须全赋值 0)，仅使用后 39 位作为地址。

RISC-V 内存管理的最小单位是$ 2^{12} = 4KB $，也就是 1 page.

页表以三级的树型结构存储在物理内存中。每个 Page Directory 大小 4KB，每个 PTE（Page Table Entry）占 64bits，一个 Page Directory 中包含 512个 PTE。

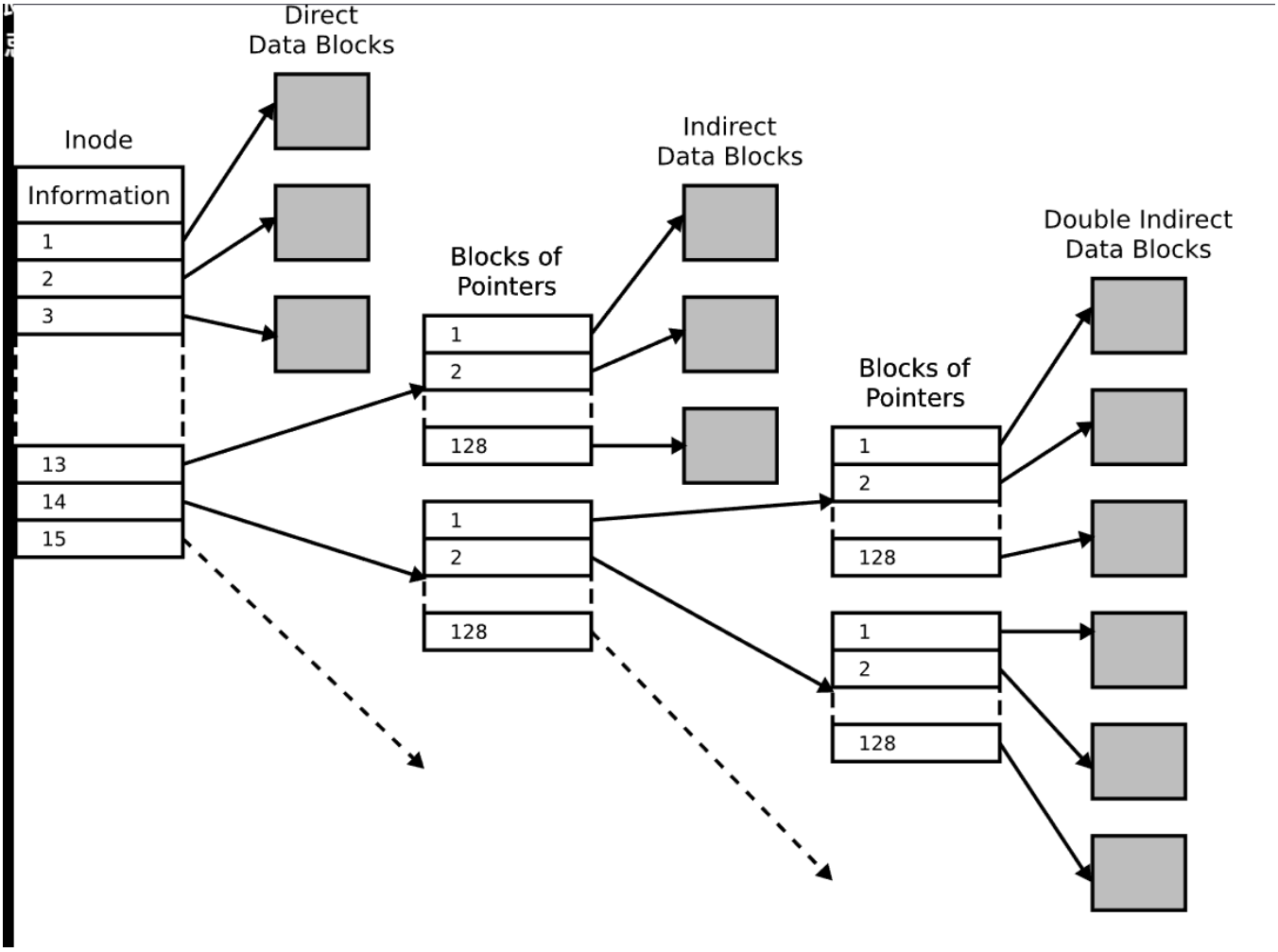
如图所示，实际的转换分三个步骤进行。注意，3 级 page table 的查找都发生在硬件中。

该树的根是一个 4KB 的 Page Directory，其中包含 512 个 PTE，每个 PTE 中包含该树下一级页表页的物理地址。

这些页中的每一个 PTE 都包含该树最后一级的 512 个 PTE（也就是说每个 PTE 占 8 个字节，正如图 3.2 最下面所描绘的）。

分页硬件使用 27 位中的前 9 位在根页表页面中选择 PTE，中间 9 位在树的下一级页表页面中选择 PTE，最后 9 位选择最终的 PTE。

整个过程就和 EXT 文件系统里的 inode 的 block 指针类似。



三级结构的优点就是更加节省内存，可以按需分配页面。举个例子，如果一个应用程序只使用了一个页面，那么顶级页面目录将只使用条目 0，条目 1 到 511 都将被忽略，因此内核不必为这 511 个条目所对应的中间页面目录分配页面，也就更不必为这 511 个中间页目录分配底层页目录的页。

缺点是 CPU 必须从内存中加载三个 PTE 以将虚拟地址转换为物理地址。为了减少从物理内存加载 PTE 的开销，RISC-V CPU 将页表条目缓存在 Translation Look-aside Buffer (TLB) 中。

最后，Page Table 十分灵活，不仅可以实现进程的地址空间隔离，也可以配合 Page Fault 异常实现许多高级的功能，比如 COW（Copy on write），mmap（memory mapped files）等等

TLB（Translation Lookaside Buffer）

你可以发现，当处理器从内存加载或者存储数据时，基本上都要做 3 次内存查找，第一次在最高级的 page directory，第二次在中间级的 page directory，最后一次在最低级的 page directory。所以对于一个虚拟内存地址的寻址，需要读三次内存，这里代价有点高。所以实际中，几乎所有的处理器都会对于最近使用过的虚拟地址的翻译结果有缓存。这个缓存被称为：Translation Lookside Buffer（通常翻译成页表缓存）。你会经常看到它的缩写 TLB。基本上来说，这就是 Page Table Entry 的缓存，也就是 PTE 的缓存。

当处理器第一次查找一个虚拟地址时，硬件通过 3 级 page table 得到最终的 PPN，TLB 会保存虚拟地址到物理地址的映射关系。这样下一次当你访问同一个虚拟地址时，处理器可以查看 TLB，TLB 会直接返回物理地址，而不需要通过 page table 得到结果。

注意：如果你切换了 page table，操作系统需要告诉处理器当前正在切换 page table，处理器会清空 TLB。因为本质上来说，如果你切换了 page table，TLB 中的缓存将不再有用，它们需要被清空，否则地址翻译可能会出错。所以操作系统知道 TLB 是存在的，但只会时不时的告诉操作系统，现在的 TLB 不能用了，因为要切换 page table 了。在 RISC-V 中，清空 TLB 的指令是 `sfence_vma`。

实验

实验介绍

实验内容 1 - Lazy allocation

```shell
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab7-1-lazy    #切换到本实验的分支
```

任务：

- 操作系统可以使用页表硬件的技巧之一是延迟分配用户空间堆内存（lazy allocation of user-space heap memory）。
- Xv6应用程序使用 `sbrk()` 系统调用向内核请求堆内存。在我们给出的内核中，`sbrk()` 分配物理内存并将其映射到进程的虚拟地址空间。内核为一个大请求分配和映射内存可能需要很长时间。例如，考虑由262144个4096字节的页组成的千兆字节；即使单独一个页面的分配开销很低，但合起来如此大的分配数量将不可忽视。
- 此外，有些程序申请分配的内存比实际使用的要多（例如，实现稀疏数组），或者为了以后的不时之需而分配内存。为了让 `sbrk()` 在这些情况下更快地完成，复杂的内核会延迟分配用户内存。
- 也就是说，`sbrk()` 不分配物理内存，只是记住分配了哪些用户地址，并在用户页表中将这些地址标记为无效。当进程第一次尝试使用延迟分配中给定的页面时，CPU 生成一个页面错误（page fault），内核通过分配物理内存、置零并添加映射来处理该错误。您将在这个实验中向 xv6添加这个延迟分配特性。

Tips：

- 修改 *trap.c* 中的代码以响应来自用户空间的页面错误，方法是新分配一个物理页面并映射到发生错误的地址，然后返回到用户空间，让进程继续执行。您应该在生成“`usertrap(): …`”消息的 `printf` 调用之前添加代码。你可以修改任何其他 xv6内核代码，以使 `echo hi` 正常工作。

实验内容 2 - COW fork

```shell
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab7-2-cow    #切换到本实验的分支
```

任务：

- 虚拟内存提供了一定程度的间接寻址：
  - 内核可以通过将 PTE 标记为无效或只读来拦截内存引用，从而导致页面错误，还可以通过修改 PTE 来更改地址的含义。
  - 在计算机系统中有一种说法，任何系统问题都可以用某种程度的抽象方法来解决。Lazy allocation 实验中提供了一个例子。
- 本实验要求你实现copy-on-write (COW) fork()

Tips：

- copy-on-write (COW) fork()的目标是推迟到子进程实际需要物理内存拷贝时再进行分配和复制物理内存页面。
- COW fork()只为子进程创建一个页表，用户内存的 PTE 指向父进程的物理页。COW fork()将父进程和子进程中的所有用户 PTE 标记为不可写。当任一进程试图写入其中一个 COW 页时，CPU 将强制产生页面错误。内核页面错误处理程序检测到这种情况将为出错进程分配一页物理内存，将原始页复制到新页中，并修改出错进程中的相关 PTE 指向新的页面，将 PTE 标记为可写。当页面错误处理程序返回时，用户进程将能够写入其页面副本。
- COW fork()将使得释放用户内存的物理页面变得更加棘手。给定的物理页可能会被多个进程的页表引用，并且只有在最后一个引用消失时才应该被释放。

拓展实验内容

> 上文实验二其实是对实验一的一种拓展，更加复杂的利用了 Lazy Allocation. 请你实现 automatically extending stack，即自动拓展进程的栈空间。当进程的栈 stack 用尽的时候，你可以仿照实验一的思路，对栈空间进行拓展。

Tips：

- https://unix.stackexchange.com/questions/63742/what-is-automatic-stack-expansion