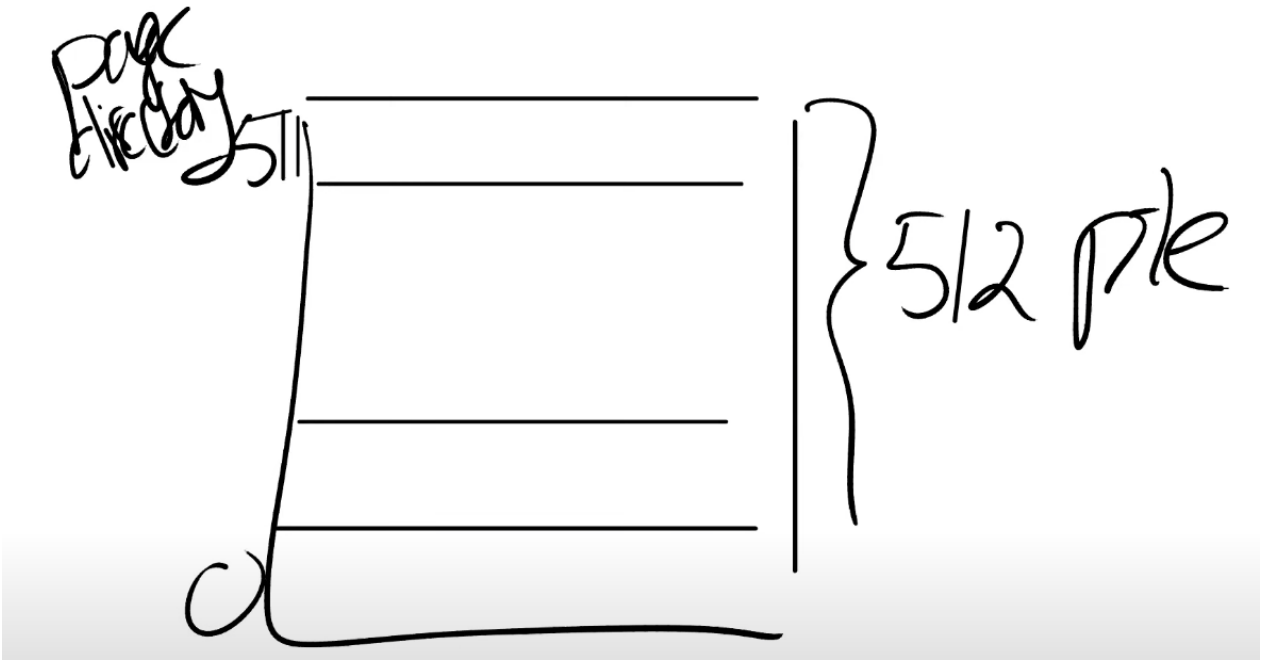
4.8 walk 函数

> 学生提问：我对于walk函数有个问题，从代码看它返回了最高级page table的PTE，但是它是怎么工作的呢？（注，应该是学生理解有误，walk函数模拟了MMU，返回的是va对应的最低级page table的PTE）

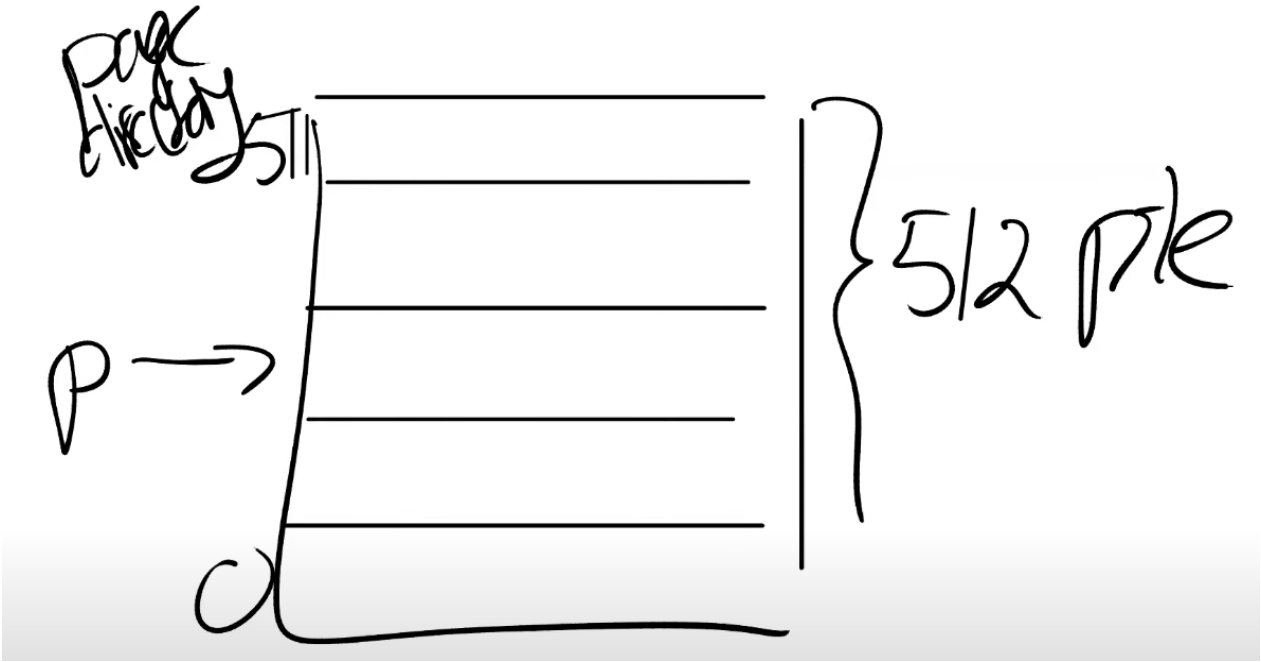
```
// Return the address of the PTE in page table pagetable
// that corresponds to virtual address va.  If alloc!=0,
// create any required page-table pages.
//
// The risc-v Sv39 scheme has three levels of page-table
// pages. A page-table page contains 512 64-bit PTEs.
// A 64-bit virtual address is split into five fields:
//   39..63 -- must be zero.
//   30..38 -- 9 bits of level-2 index.
//   21..29 -- 9 bits of level-1 index.
//   12..20 -- 9 bits of level-0 index.
//    0..11 -- 12 bits of byte offset within the page.
pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
  if(va >= MAXVA)
    panic("walk");

  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
}
```

> Frans教授：这个函数会返回page table的PTE，而内核可以读写PTE。我来画个图，首先我们有一个page directory，这个page directory 有512个PTE。最下面是0，最上面是511。



这个函数的作用是返回某一个PTE的指针。



这是个虚拟地址，它指向了这个PTE。之后内核可以通过向这个地址写数据来操纵这条PTE执行的物理page。当page table被加载到SATP寄存器，这里的更改就会生效。

从代码看，这个函数从level2走到level1然后到level0，如果参数alloc不为0，且某一个level的page table不存在，这个函数会创建一个临时的page table，将内容初始化为0，并继续运行。所以最后总是返回的是最低一级的page directory的PTE。

如果参数alloc没有设置，那么在第一个PTE对应的下一级page table不存在时就会返回。

学生提问：对于walk函数，我有一个比较困惑的地方，在写完SATP寄存器之后，内核还能直接访问物理地址吗？在代码里面看起来像是通过page table将虚拟地址翻译成了物理地址，但是这个时候SATP已经被设置了，得到的物理地址不会被认为是虚拟地址吗？

Frans教授：让我们来看kvminithart函数，这里的kernel_page_table是一个物理地址，并写入到SATP寄存器中。从那以后，我们的代码运行在一个我们构建出来的地址空间中。在之前的kvminit函数中，kvmmap会对每个地址或者每个page调用walk函数。所以你的问题是什么？

学生：我想知道，在SATP寄存器设置完之后，walk是不是还是按照相同的方式工作？

Frans：是的。它还能工作的原因是，内核设置了虚拟地址等于物理地址的映射关系，这里很重要，因为很多地方能工作的原因都是因为内核设置的地址映射关系是相同的。

学生：每一个进程的SATP寄存器存在哪？

Frans：每个CPU核只有一个SATP寄存器，但是在每个proc结构体，如果你查看proc.h，里面有一个指向page table的指针，这对应了进程的根page table物理内存地址。

```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  struct proc *parent;         // Parent process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

学生提问：为什么通过3级page table会比一个超大的page table更好呢？

Frans教授：这是个好问题，这的原因是，3级page table中，大量的PTE都可以不存储。比如，对于最高级的page table里面，如果一个PTE为空，那么你就完全不用创建它对应的中间级和最底层page table，以及里面的PTE。所以，这就是像是在整个虚拟地址空间中的一大段地址完全不需要有映射一样。

> 学生：所以3级page table就像是按需分配这些映射块。
>
> Frans教授：是的，就像前面（4.6）介绍的一样。最开始你只有3个page table，一个是最高级，一个是中间级，一个是最低级的。随着代码的运行，我们会创建更多的page table diretory。