vm.c，exec.c 阅读源码

https://blog.csdn.net/zzy980511/article/details/129912497

vm.c

函数	使用说明	页对齐
`pte_t * walk(pagetable_t pagetable, uint64 va, int alloc)`	用**软件来模拟硬件**MMU查找页表的过程，返回以pagetable为根页表，经过多级索引之后va这个虚拟地址所对应的**页表项的地址**	不需要
`int mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)`	装载一个新的映射	不需要
`uint64 walkaddr(pagetable_t pagetable, uint64 va)`	查找**用户页表**中特定虚拟地址va所对应的**物理地址**。	不需要
`void freewalk(pagetable_t pagetable)`	回收页表页的内存，保证**叶子级别页表的映射关系全部解除并释放**	
`void uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)`	取消**用户进程页表**中指定范围的映射关系	需要
`uint64 uvmdealloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz)`	回收用户页	不需要
`uint64 uvmalloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz)`	分配用户页	不需要
`int uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)`	将父进程的整个地址空间全部复制到子进程中，这包括页表本身和页表指向的物理内存中的数据。	

walk

用**软件来模拟硬件**MMU查找页表的过程，返回以pagetable为根页表，经过多级索引之后va这个虚拟地址所对应的**页表项的地址**，如果alloc != 0，则在需要时创建新的页表页，反之则不用。

walk函数返回0，只有一种情况，那就是某一级页表页在查询时发现不存在

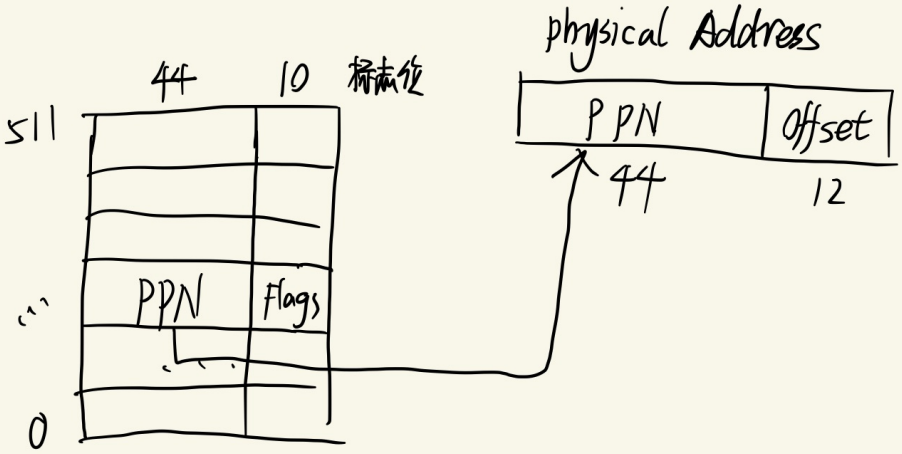
walk 函数

e 用软件模拟硬件 MMU 查找页表的过程.

返回 va 这个虚拟地址对应的页表项

PA2PTE(pa) ((((uint64) pa) >>12) <<10

PTE2PA(pte) (((pte) >> 10) <<12




```
pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc)
{
  if (va >= MAXVA)
    panic("walk");

  // 这里只走了两层，最后pagetable是叶子页表
  for (int level = 2; level > 0; level--)
  {
    //pagetable 其实可以理解为一个数组
    //pagetable[PX(level, va)] 取出了其中的值
    //&pagetable[PX(level, va)] 取出它的地址
    //用pte_t * 接收它的地址
```

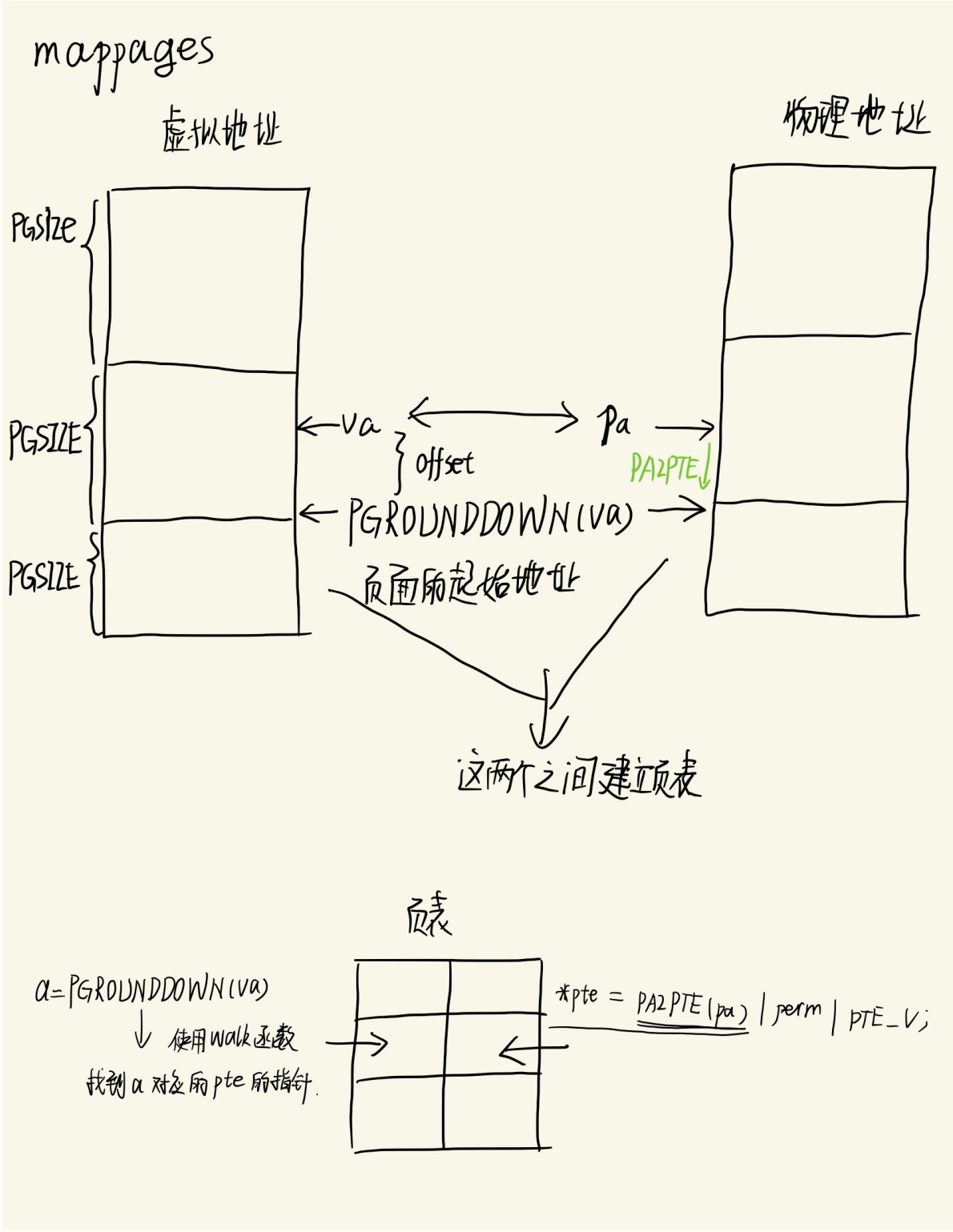
```
    //pte 是 地址
    //*pte 是 具体的值 也就是 pagetable[PX(level, va)]
    pte_t *pte = &pagetable[PX(level, va)];
    if (*pte & PTE_V)
    {
      // 这行代码从PTE中提取出物理地址，直接赋值给pagetable指针(而它是一个虚拟地址)
      // 这样赋值合理吗？只有在虚拟地址==物理地址时合理，即直接映射。
      pagetable = (pagetable_t)PTE2PA(*pte);
    }
    else
    {
      if (!alloc || (pagetable = (pde_t *)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
}
```

mappages

这个函数是用来装载一个新的映射

```
// PGROUNDDOWN(a)：地址a所在页面是多少号页面，拉回所在页面开始地址
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
```

给你一个虚拟地址，和一个物理地址，把它们两拼在一起，放在页表里面




```c
int mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
{
  // a存储的是当前虚拟地址对应的页
  // last存放的是最后一个应设置的页
  // 当 a==last时，表示a已经设置完了所有页，完成了所有任务
  uint64 a, last;
  pte_t *pte;

  // a,last向下取整到页面开始位置，设置last相当于提前设置好了终点页
  a = PGROUNDDOWN(va);
  last = PGROUNDDOWN(va + size - 1);
  for (;;)
  {
```

```
    if ((pte = walk(pagetable, a, 1)) == 0)
      return -1;
    if (*pte & PTE_V)
      panic("remap");
    *pte = PA2PTE(pa) | perm | PTE_V;
    if (a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

kvminit/kvminithart

```
// 将trampline页面映射到内核虚拟地址空间的最高一个页面
// TRAMPOLINE的定义如下，就是最高虚拟地址减去一个页面大小
// #define TRAMPOLINE (MAXVA - PGSIZE)
kvmmap(TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
```

在kvminit中其实是还没有分页机制的，在kvminithart中才开始有了分页机制

```
void kvminithart()
{
  w_satp(MAKE_SATP(kernel_pagetable));
  sfence_vma();
}
```

自此之后虚拟地址**就要经过MMU的翻译**才可以转化为物理地址了

所以说前面的walk是模拟MMU来着，还没有开始用MMU呢，在kvminithart之后才开始用MMU

walkaddr

专门用来查找用户页表中特定虚拟地址va所对应的物理地址。

1.它只用来查找用户页表
2.返回的是物理地址，而非像walk函数那样只返回最终层的PTE的指针

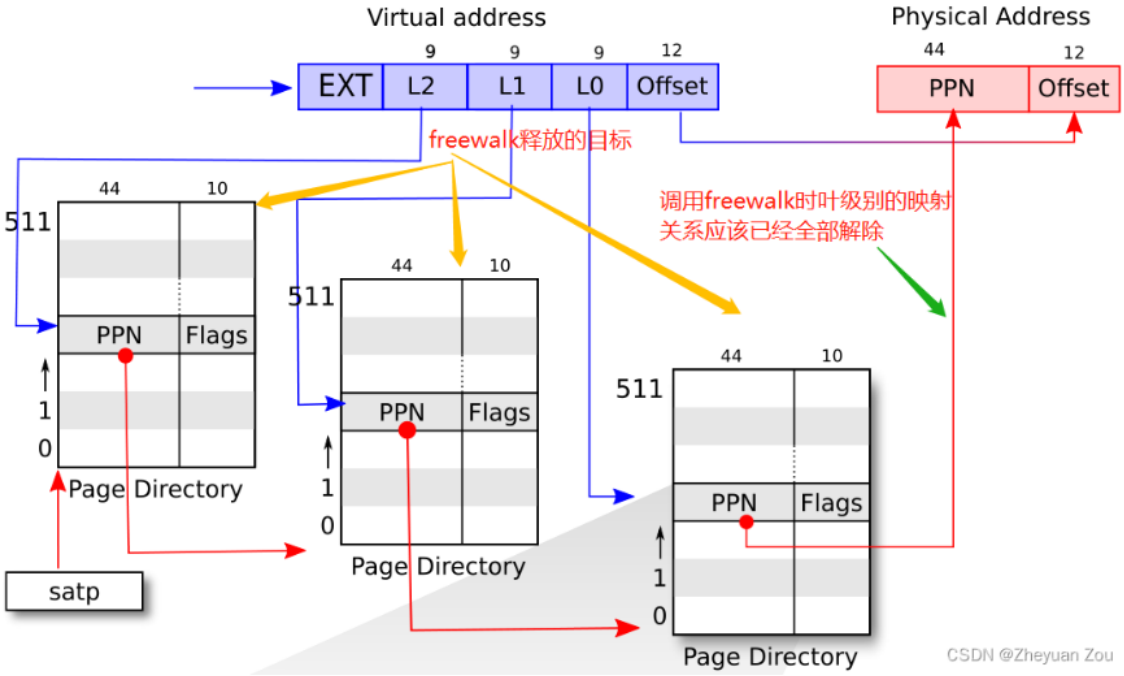
```
uint64 walkaddr(pagetable_t pagetable, uint64 va)
{
  pte_t *pte;
  uint64 pa;

  if (va >= MAXVA)
    return 0;

  pte = walk(pagetable, va, 0);
  if (pte == 0)
    return 0;
  if ((*pte & PTE_V) == 0)
    return 0;
  if ((*pte & PTE_U) == 0)
    return 0;
```

```
  pa = PTE2PA(*pte);
  return pa;
}
```

freewalk

专门用来回收页表页的内存的

在调用这个函数时应该保证**叶子级别页表的映射关系全部解除并释放**(这将会由后面的uvmunmap函数负责)




```
void freewalk(pagetable_t pagetable)
{
  // there are 2^9 = 512 PTEs in a page table.
  for (int i = 0; i < 512; i++)
  {
    pte_t pte = pagetable[i];
    //通过标志位判断是否到达了叶子页表
    //如果有效位为1，并且读/写/可执行位都是0
    //说明这不是叶子页表
    if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0)
    {
      // this PTE points to a lower-level page table.
      uint64 child = PTE2PA(pte);
      freewalk((pagetable_t)child);
      pagetable[i] = 0;
    }
    else if (pte & PTE_V)
    {
      //表示这是一个叶级PTE，且未经释放，这不符合本函数调用条件，会陷入一个panic
      panic("freewalk: leaf");
    }
  }
  //最后释放页表本身占用的内存
```

```
  kfree((void *)pagetable);
}
```

uvm

uvminit

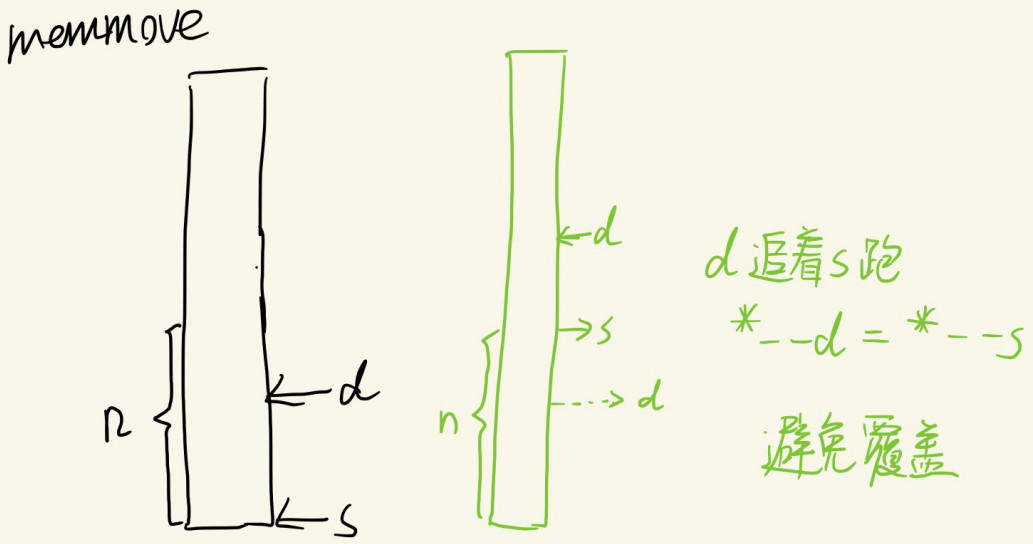
```
246  // a user program that calls exec("/init")
247  // od -t xC initcode
248  uchar initcode[] = {       Robert Morris, 4 years ago • cle
249      0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x45, 0x02,
250      0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x35, 0x02,
251      0x93, 0x08, 0x70, 0x00, 0x73, 0x00, 0x00, 0x00,
252      0x93, 0x08, 0x20, 0x00, 0x73, 0x00, 0x00, 0x00,
253      0xef, 0xf0, 0x9f, 0xff, 0x2f, 0x69, 0x6e, 0x69,
254      0x74, 0x00, 0x00, 0x24, 0x00, 0x00, 0x00, 0x00,
255      0x00, 0x00, 0x00, 0x00};
256
257  // Set up first user process.
258  void userinit(void)
259  {
260    struct proc *p;
261
262    p = allocproc();
263    initproc = p;
264
265    // allocate one user page and copy init's instructions
266    // and data into it.
267    uvminit(p->pagetable, initcode, sizeof(initcode));
268    p->sz = PGSIZE;
269
```

这个函数是针对操作系统第一个进程而言的，这个函数的作用就是将initcode映射到用户地址空间去，从虚拟地址0开始

```
void uvminit(pagetable_t pagetable, uchar *src, uint sz)
{
  char *mem;

  if (sz >= PGSIZE)
    panic("inituvm: more than a page");
  //先申请一片物理空间存放initcode
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  //把这片物理空间和虚拟地址0对应起来
  mappages(pagetable, 0, PGSIZE, (uint64)mem, PTE_W | PTE_R | PTE_X | PTE_U);
  //把initcode放入这片物理空间中
  memmove(mem, src, sz);
}
```

memmove

`void* memmove(void *dst, const void *src, uint n)` 实现从src拷贝n个字节到dst去




```
void*
memmove(void *dst, const void *src, uint n)
{
  const char *s;
  char *d;

  s = src;
  d = dst;
  if(s < d && s + n > d){
    s += n;
```

```
    d += n;
    while(n-- > 0)
      *--d = *--s;
  } else
    while(n-- > 0)
      *d++ = *s++;

  return dst;
}
```

如何避免覆盖，这是一段很神奇的代码

uvmunmap

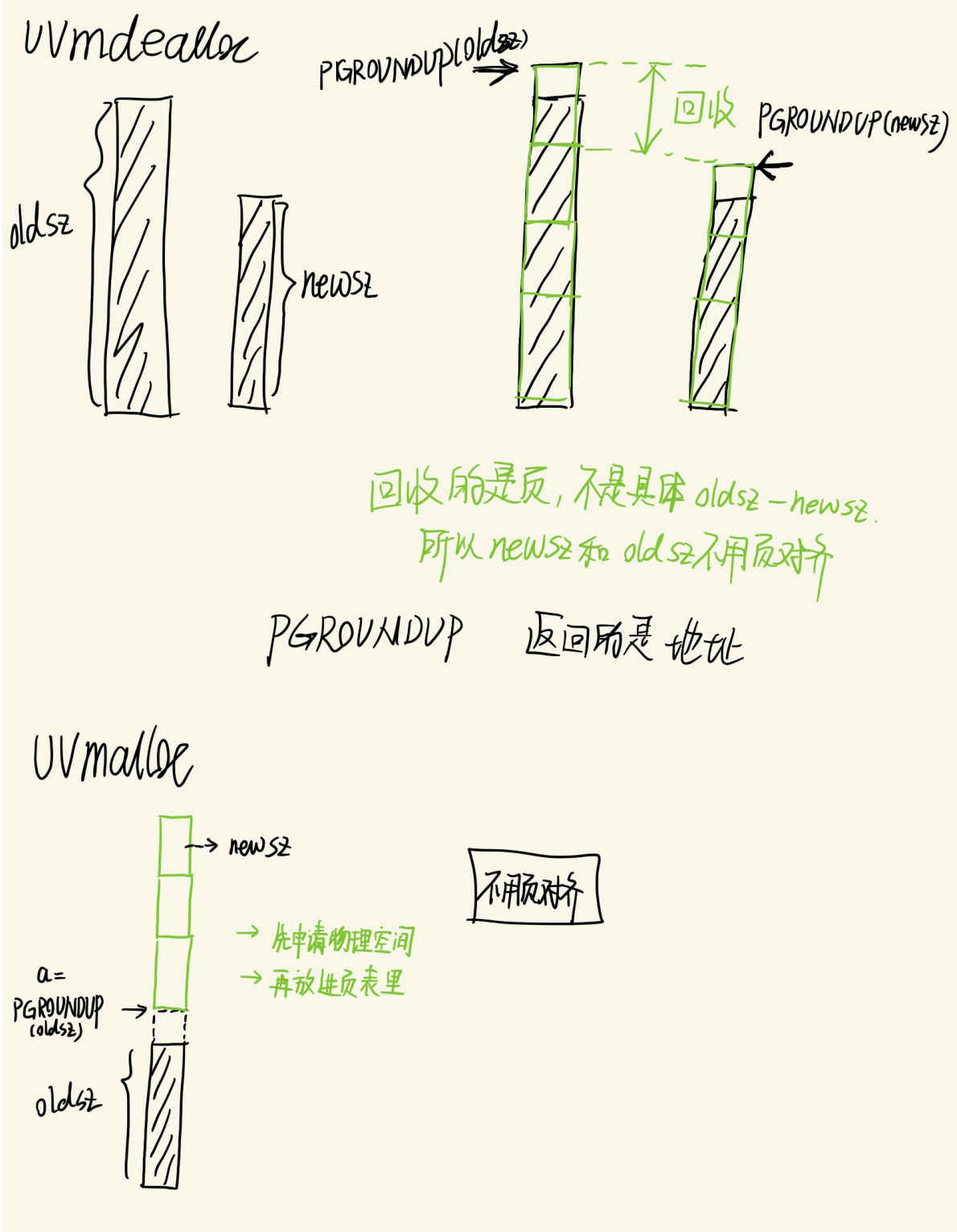
取消用户进程页表中指定范围的映射关系

`void uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)`从虚拟地址va开始释放npages个页面，但是要注意va一定要是页对齐的。如果设置标志位do_free，此函数还会一并将分配出去的物理页面也进行回收。

```
void uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
  uint64 a;
  pte_t *pte;

//不是页对齐的，陷入panic
  if ((va % PGSIZE) != 0)
    panic("uvmunmap: not aligned");

  for (a = va; a < va + npages * PGSIZE; a += PGSIZE)
  {
    // 如果虚拟地址在索引过程中对应的中间页表页不存在，陷入panic
    if ((pte = walk(pagetable, a, 0)) == 0)
      panic("uvmunmap: walk");
    // 查找成功，但发现此PTE不存在，陷入panic
    if ((*pte & PTE_V) == 0)
      panic("uvmunmap: not mapped");
    // 查找成功，但发现此PTE除了valid位有效外，其他位均为0
    if (PTE_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    // 如果do_free被置位，那么还要释放掉PTE对应的物理内存
    if (do_free)
    {
      uint64 pa = PTE2PA(*pte);
      kfree((void *)pa);
    }
    // 最后将PTE本身全部清空，成功解除了映射关系
    *pte = 0;
  }
}
```

uvmdealloc

回收用户页



`uint64 uvmdealloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz)` 使得内存大小从oldzd变成newsz

```
// PGROUNDUP(sz)：sz大小的内存至少使用多少页才可以存下，返回的是下一个未使用页的地址
#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
```

```
uint64 uvmdealloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz)
{
  // 如果新的内存大小比原先内存还要大，那么什么也不用做，直接返回oldsz即可
```

```
  if (newsz >= oldsz)
    return oldsz;

  if (PGROUNDUP(newsz) < PGROUNDUP(oldsz))
  {
    int npages = (PGROUNDUP(oldsz) - PGROUNDUP(newsz)) / PGSIZE;
    uvmunmap(pagetable, PGROUNDUP(newsz), npages, 1);
  }

  return newsz;
}
```

uvmalloc

分配用户页

```
uint64 uvmalloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz)
{
  char *mem;
  uint64 a;

  if (newsz < oldsz)
    return oldsz;

  oldsz = PGROUNDUP(oldsz);
  for (a = oldsz; a < newsz; a += PGSIZE)
  {
    // 如果mem为空指针，表示内存耗尽
    // 释放之前分配的所有内存，返回0表示出错
    mem = kalloc();
    if (mem == 0)
    {
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if (mappages(pagetable, a, PGSIZE, (uint64)mem, PTE_W | PTE_X | PTE_R |
PTE_U) != 0)
    {
      kfree(mem);
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
  }
  return newsz;
}
```

uvmcopy

将父进程的整个地址空间全部复制到子进程中，这包括页表本身和页表指向的物理内存中的数据。

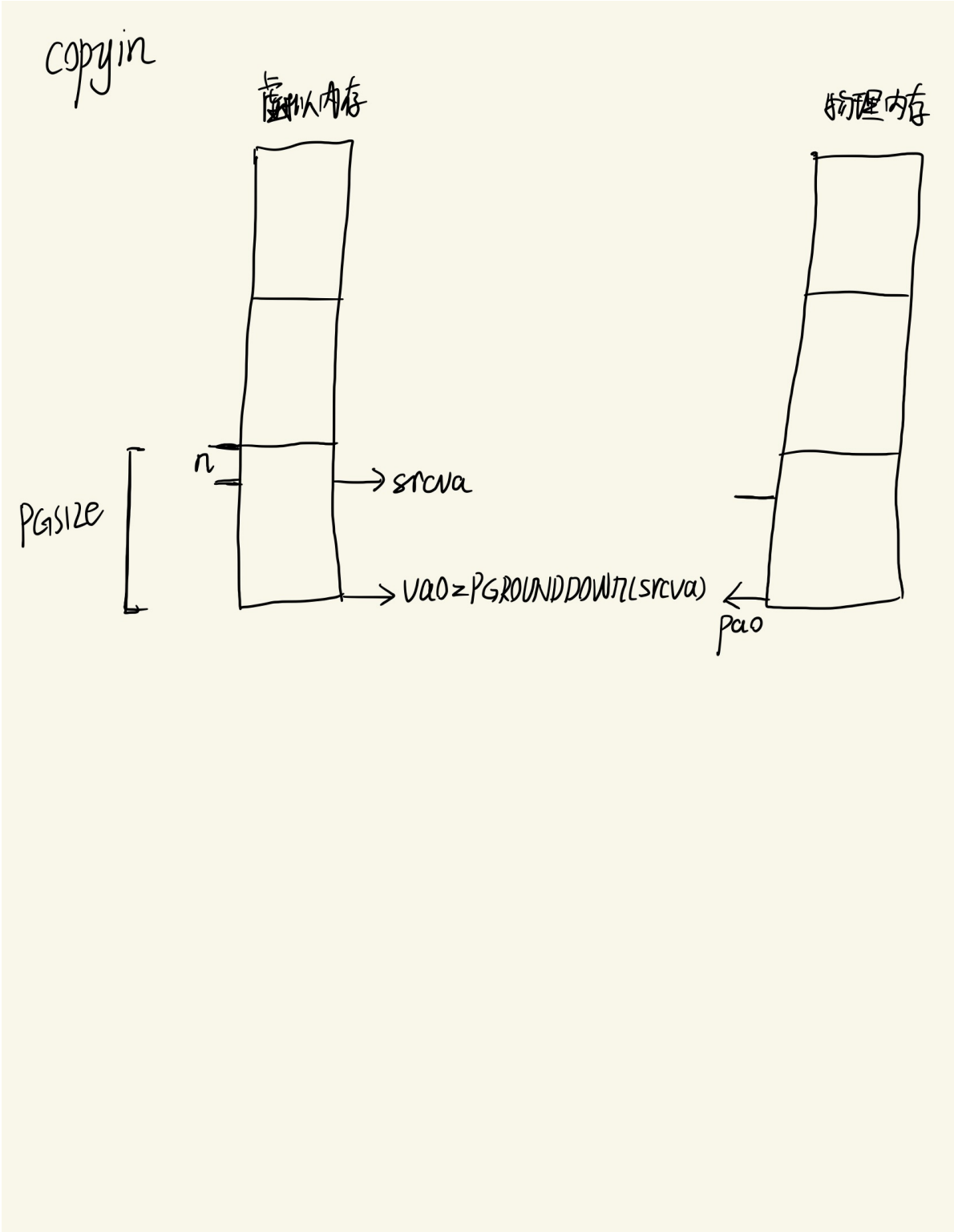
用户页表不存在没有使用过的页

自下而上：text、data、guard page、stack、heap

```c
int uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  char *mem;

// sz指要复制的地址空间大小，被调用时传入p->sz，表示整个地址空间
  // 对整个地址空间逐页复制
  for (i = 0; i < sz; i += PGSIZE)
  {
    // 如果寻找过程中发现有PTE不存在，陷入panic
    if ((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    // PTE存在但是对应页未被使用，陷入panic
    // 再次强调，用户空间的内存使用是严格紧密的，中间不会有未使用的页存在
    // 自下而上：text、data、guard page、stack、heap
    if ((*pte & PTE_V) == 0)
      panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if ((mem = kalloc()) == 0)
      goto err;
    memmove(mem, (char *)pa, PGSIZE);
    if (mappages(new, i, PGSIZE, (uint64)mem, flags) != 0)
    {
      kfree(mem);
      goto err;
    }
  }
  return 0;

err:
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
}
```

copyin

`int copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)` 从用户页表pagetable的虚拟地址srcva处拷贝长度为len字节的数据到地址dst



在内核状态下，凡是指针变量，都会通过MMU硬件查询内核页表变成对应的物理地址。

用户态下的虚拟地址，就无法使用MMU来翻译了，只能用软件模拟。

```
int copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
  uint64 n, va0, pa0;

  while (len > 0)
  {
    va0 = PGROUNDDOWN(srcva);
    pa0 = walkaddr(pagetable, va0);
    if (pa0 == 0)
      return -1;
```

```
    //如果不是页对齐，则复制第一个页面剩余的字节数量
    //如果是页对齐，则复制整个页面
    n = PGSIZE - (srcva - va0);
    if (n > len)
      n = len;
    memmove(dst, (void *)(pa0 + (srcva - va0)), n);

    len -= n;
    dst += n;
    srcva = va0 + PGSIZE;
  }
  return 0;
}
```

copyinstr

因为copyinstr复制的对象是字符串，所以复制结束的标志是遇到空字符(null-terminated)。

为了防止缓冲区的溢出，还特地设置了最大可复制的字符数量上限max，如果超过这个数字仍未遇到空字符，则要退出复制过程并返回-1表示出错了。

```
int copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
{
  uint64 n, va0, pa0;
  int got_null = 0;

  //是否到了空字符或者超出了max长度
  while (got_null == 0 && max > 0)
  {
    va0 = PGROUNDDOWN(srcva);
    pa0 = walkaddr(pagetable, va0);
    if (pa0 == 0)
      return -1;
    n = PGSIZE - (srcva - va0);
    if (n > max)
      n = max;

    char *p = (char *)(pa0 + (srcva - va0));
    //上面代码和copyin很像
    //这里复制n个字符
    while (n > 0)
    {
      //遇到空字符就退出
      if (*p == '\0')
      {
        *dst = '\0';
        got_null = 1;
        break;
      }
      else
      {
        *dst = *p;
      }
      --n;
      --max;
```

```
      p++;
      dst++;
    }

    srcva = va0 + PGSIZE;
  }
  if (got_null)
  {
    return 0;
  }
  else
  {
    return -1;
  }
}
```

copyout

这个跟上面的很像

就是用户内存的虚拟地址，需要一些函数转化成物理地址

但是内核的虚拟地址不需要，可以硬件直接转化。

```
int copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;

  while (len > 0)
  {
    va0 = PGROUNDDOWN(dstva);
    pa0 = walkaddr(pagetable, va0);
    if (pa0 == 0)
      return -1;
    n = PGSIZE - (dstva - va0);
    if (n > len)
      n = len;
    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }
  return 0;
}
```

exec.c