3.page table

Tags

xv6 book

Xv6 runs on Sv39 : only the bottom 39 bits of a 64-bit virtual address are used

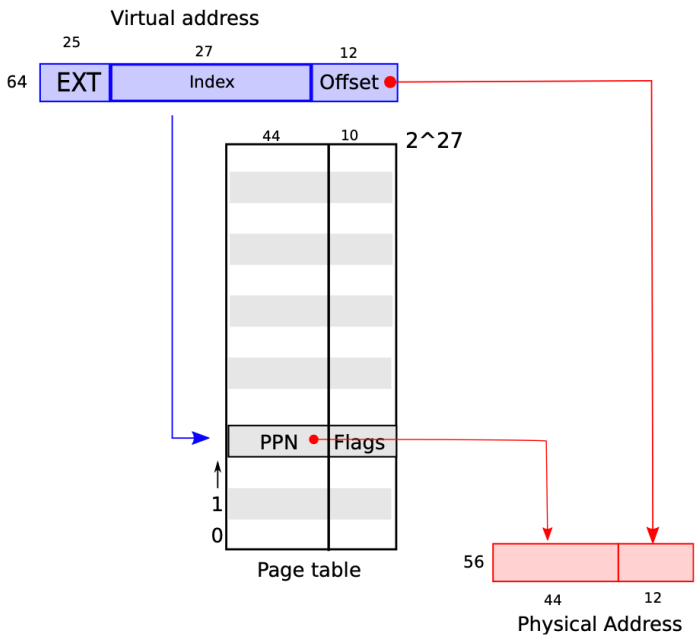
a RISC-V page table is logically an array of 2^27 (134,217,728) *page table entries (PTEs)*.

Each PTE contains a 44-bit physical page number (PPN) and some flags

translation a virtual address by using the **top 27 bits** to index into the page table to find a **PTE**,

get a 56-bit physical address : **top 44 bits** from the PPN

both vir and phy : **bottom 12 bits** are common used.(27+12=39,44+12=56)



top 25 bits of a virtual address are not used for translation

PTE for the physical page number to grow also by another 10 bits.

why this running space (2^39)for application and (2^56)for whole memory

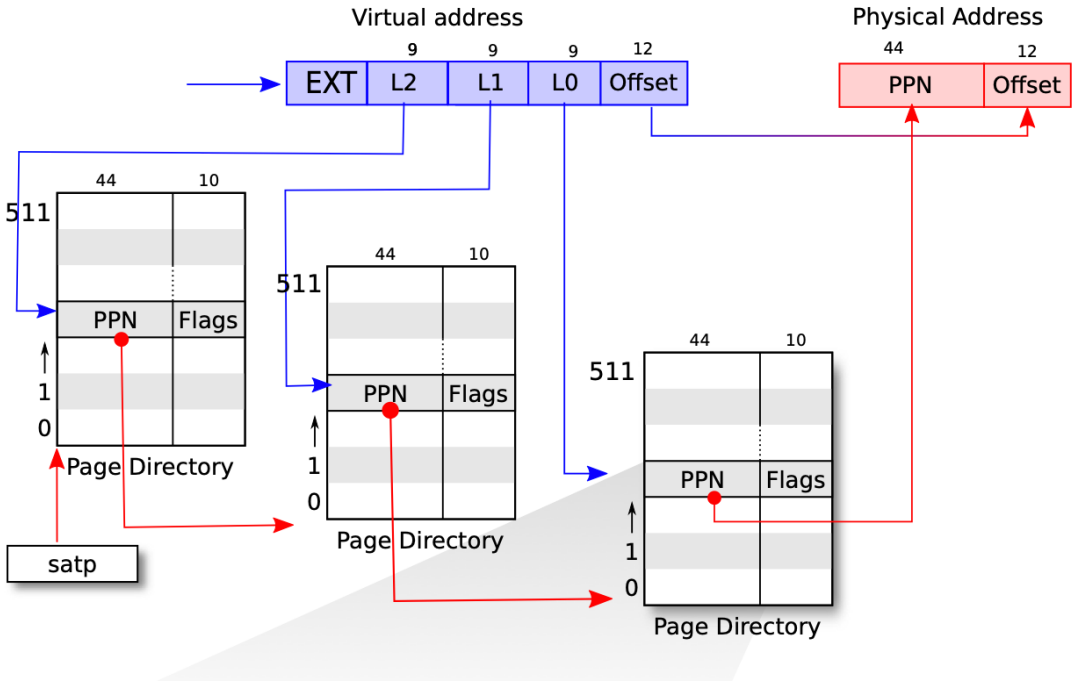
1. enough for one app running.
2. enough physical memory space for the near future to fit many I/O devices and DRAM chips.

designers have defined Sv48 with 48-bit virtual addresses.

translation process

A page table is stored in physical memory as a three-level tree.

9*3 = 27

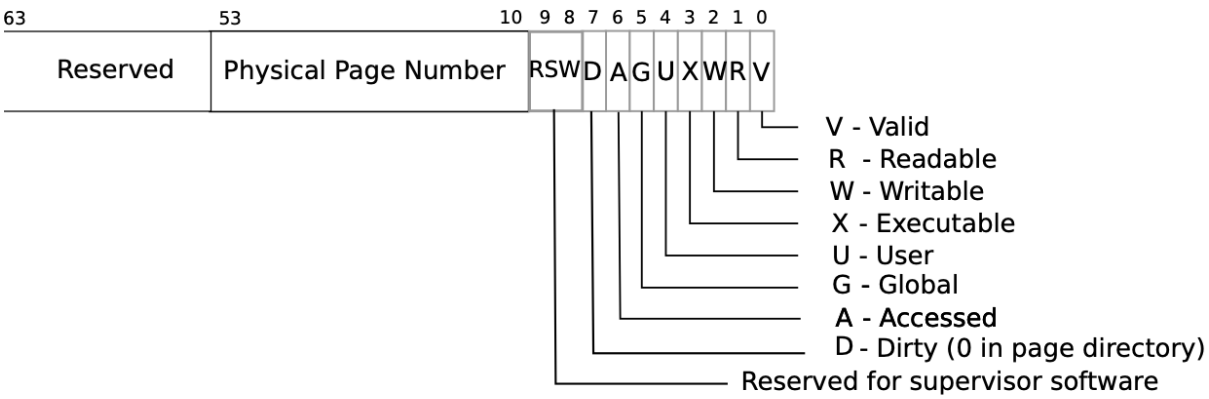


advantage: **save space** the kernel doesn't have to allocate pages those for page directories.

downside: more read and store instruction.

solution: *Translation Look-aside Buffer (TLB)*.

flag bit



flag and hardware related sections all in kernel/riscv.h

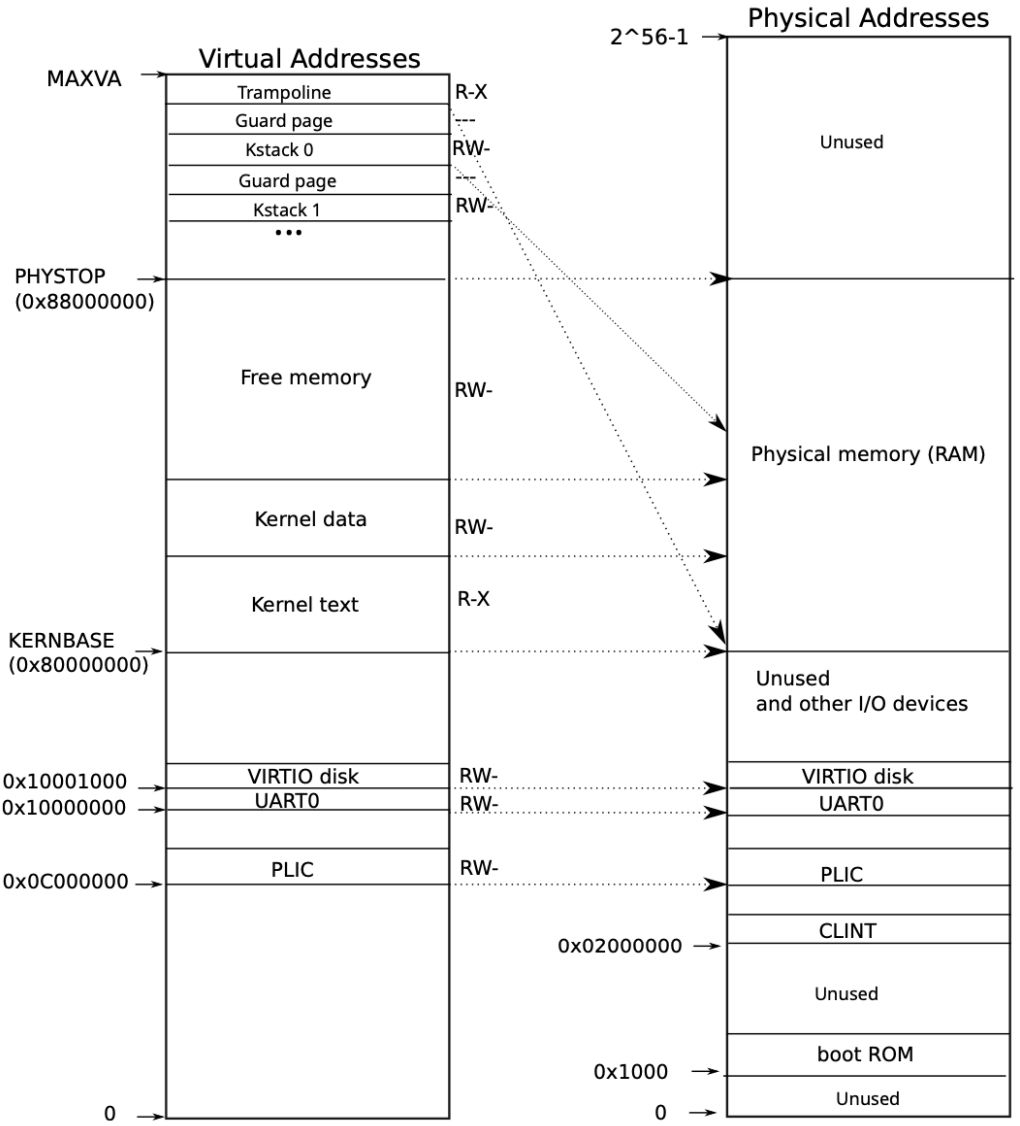
satp — root page address

Typically a kernel maps all of physical memory into its page tableland can change content of PTE.

It use standard load/store instructions , and will not be denied.

Only a part of virtual space refer to RAM

other space refer to other hardware



0-0x80000000 not real storage space. like: PLIC, UART, VIRTIO disk. (MMIO mapping to this device)

0x80000000 KERNBASE : kernel start :text, data. (RAM)

…. free memory. (RAM)

0x88000000 PHYSTOP : physical memory stop.

Trampoline, guard page, kstack0…x, … (RAM)

MAXVA

kernel address space

About qemu

simulate a conputer : RAM(physical memory) 0x80000000 — 0x88000000(PHSYSTOP) (at least)

kernel communicate with RAM and device mapping on address using ‘direct mapping’, that means these resources on the same address in both virtual and physical.

this machinsm is convenient for kernel to read/write (fork’s memory copy is directly using physical memory)

Not directly mapped memory in kernel

- trampoline page : an interesting use in kerne. and it’s mapped twice: virtual top, directly mapped.
- kernel stack. It’s each process’s independent part. It’s been mapped high and PTE can set some invalided entries to protect kernel’s memory(make referable panic).

code

pageable_t structure of RISCV

central function: walk() find PTE in virtual address;

kvm—, manipulate the kernel page table.

uvm—, manipulate the user page table.

copyin/out , for both, and need to find physical address.

kvminit early booting , create a page table

kvmmake also , alloc physical page for root page table

kvmmap install all the virtual : kernel, memory, device.

kvminithart set satp register for root page table address.

> TLB entry for each RISCV CPU , and as kernel change the PTE, it need to tell TLB to delete the corresponding one.(otherwise it will overwrite the old physical space that has been allocate to another PTE)
>
> RISCV has sfence.vma instruction to flush the TLB. it also show up in kvminithar, after set satp.(also show up at trampoline code , in which process will turn to user mode, and switch to user memory space).

mappages divide the virtual memory space at page interval. for each it call walk to find PTE, and initialize it.

walk() just like RISCV page hardware to find the page of virtual address.if the final PTE hasn’t been allocate, then walk() will allocate a new one.

proc_mapstack allocate all kernel stack. and kvmmap also map guard page for these.

physical memory allocation

must between kernel end- and PHSYSTOP.

each allocation is a 4096 whole page.

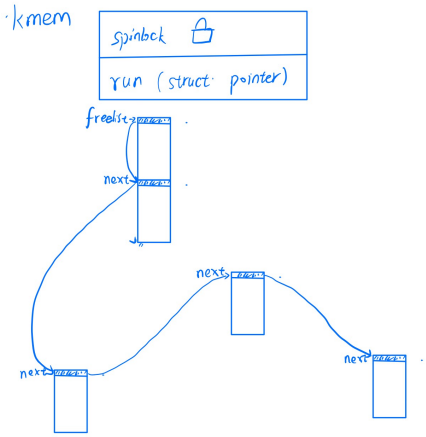
xv6 also make a free space list to enroll all free space.

(for most, it have to parse the configeration from hardware . but xv6 assume that it has 128M bytes)

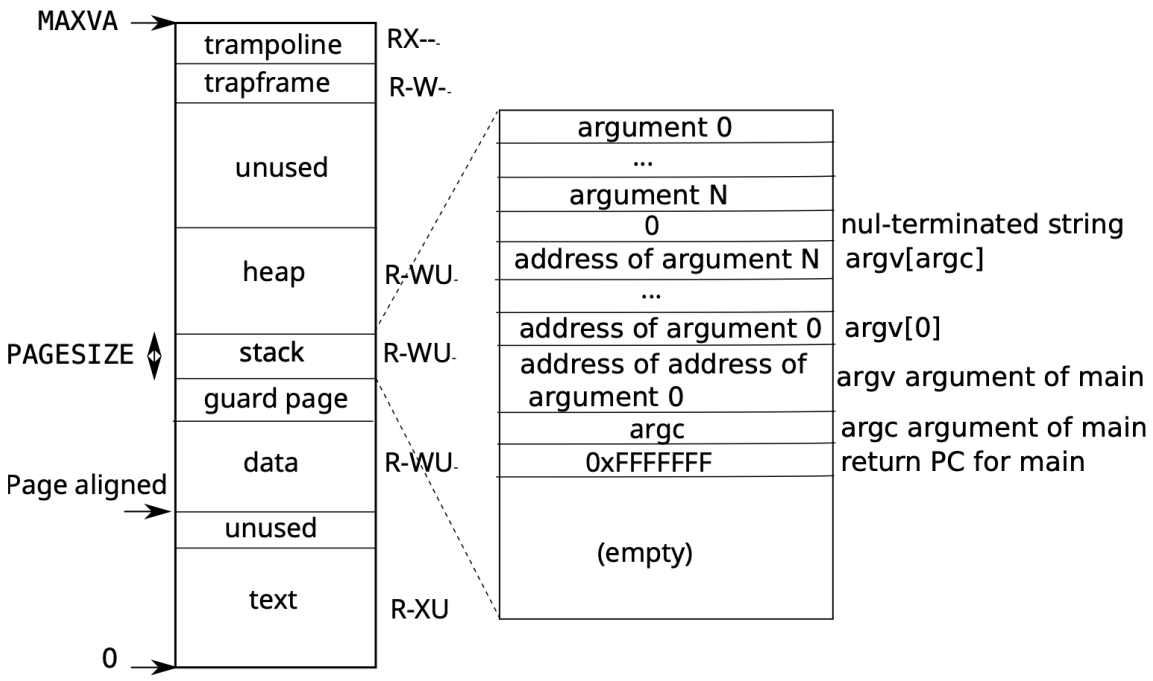
code

kinit hold every pagan a list

kalloc.c



process memory space



Each process has a separate page table.

user memory starts at virtual address zero and can grow up to MAXVA. (256 G bytes)

for user program, system more serious, and not permit it to change the text segment, which will cause page fault.(aso execute an unexecutable segment)

(The kernel then kills the process and prints informative so that the developer can track down the problem.)

for stack overflow

xv6 places an inaccessible guard page right below the stack by clearing the PTE_U flag. And also generate a **page-fault** exception.

but in this condition , system usually allocate more page for stack.

code : sbrk

for a process to shrink or grow its memory.

The system call is implemented by the function growproc. which calls uvmalloc or uvmdealloc.

alloc use kalloc . dealloc use uvmunmap(), then call walk and kfree().

code : exec

replaces a process's **user address space** with data read **from a file**

1. opens the named binary path using namei
2. read ELF (An ELF binary starts with the four-byte "magic number" 0x7F, 'E', 'L', 'F', or ELF_MAGIC)
3. alloc new page: proc_pagetable allocate new memory : uvmalloc
4. loads each segment into memory with loadseg (which use walkaddr and read)
5. note: **filesz** may be less than the **memsz**, indicating that the gap between them **should be filled with zeroes**
6. stack : allocates and initializes the user stack. It allocates just one stack page
7. check:like an invalid program segment, it jumps to the label bad, frees the new image, and returns -1
8. Once the image is complete, exec can commit to the new page table and free the old one
9. Xv6 performs a number of checks to avoid these risks
10. loadseg loads into the process's page table, not in the kernel's page table.

code review for mapping part

kvminit

initialize the kernel's pagetable

```
void
kvminit(void)
{
  kernel_pagetable = kvmmake();
}
```

use the kvmmake()

kvmmake()

make a direct-map page table for kernel

```
kvmmake(void)
{
  pagetable_t kpgtbl;

  kpgtbl = (pagetable_t) kalloc();
  memset(kpgtbl, 0, PGSIZE);

  // uart registers
  kvmmap(kpgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);

  // virtio mmio disk interface
  kvmmap(kpgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

  // PLIC
  kvmmap(kpgtbl, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
```

pagetable_t is a pointer

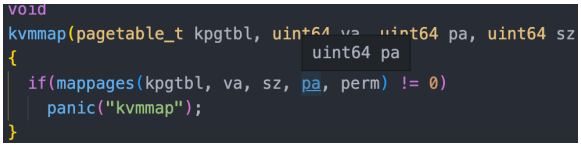
use kalloc (kernel alloc) to get a physical space address, that can put a kernel page on that.

```
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
void *
kalloc(void)
{
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

(k*** is in kernel mode , so it only use physical address, it’s easier)

kvmmap

straightly initialize the range of virtual space with kpgtbl and perm(mode) and most important — the physical address



it use mappages.(which is commonly used)

mappages

do common mapping work

while (virtual space has been all mapped)

first use walk to get the PTE correspond to the address a(can’t alloc)

if valid , panic (because it still map to another page)

fill this entry.

walk

get PTE , with va.

```
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)];
}
```

level 2-1-0, to get the final PTE

alloc can permit it to kalloc→memset(0),pte = new page.

kvminithart

h/w page table register to kernel's

```
// and enable paging.
void
kvminithart()
{
  // wait for any previous writes to the pa
  sfence_vma();

  w_satp(MAKE_SATP(kernel_pagetable));

  // flush stale entries from the TLB.
  sfence_vma();
}
```

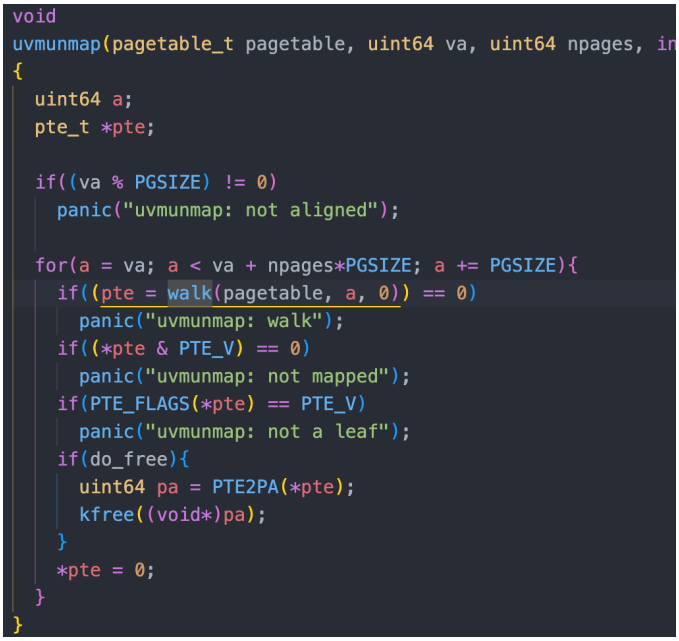
sfence.vma totally a machine instruction.

set satp

sfence.vma again

uvmunmap

usermode remove n pages.



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

  if((va % PGSIZE) != 0)
    panic("uvmunmap: not aligned");

  for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0)
      panic("uvmunmap: walk");
    if((*pte & PTE_V) == 0)
      panic("uvmunmap: not mapped");
    if(PTE_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    if(do_free){
      uint64 pa = PTE2PA(*pte);
      kfree((void*)pa);
    }
    *pte = 0;
  }
}
```

if not valid , we can't delete it.

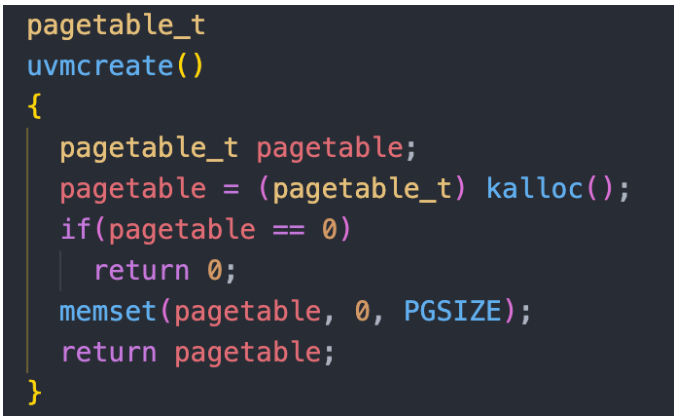
if only valid , it should be a middle level page

if do_free, it will entirely remove it , and use kfree to fill it’s content with junk.

otherwise, it remove it by set *pte.

uvmcreate

in user mode, we also need to build up page table



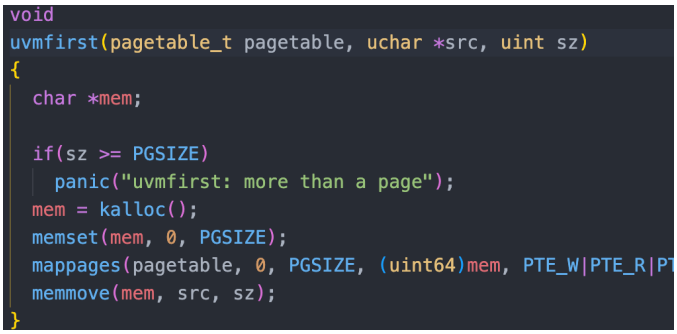
```
pagetable_t
uvmcreate()
{
  pagetable_t pagetable;
  pagetable = (pagetable_t) kalloc();
  if(pagetable == 0)
    return 0;
  memset(pagetable, 0, PGSIZE);
  return pagetable;
}
```

use kalloc and set 0 and return.

no other action.

uvmfirst

alloc and load the 0 page for user mode



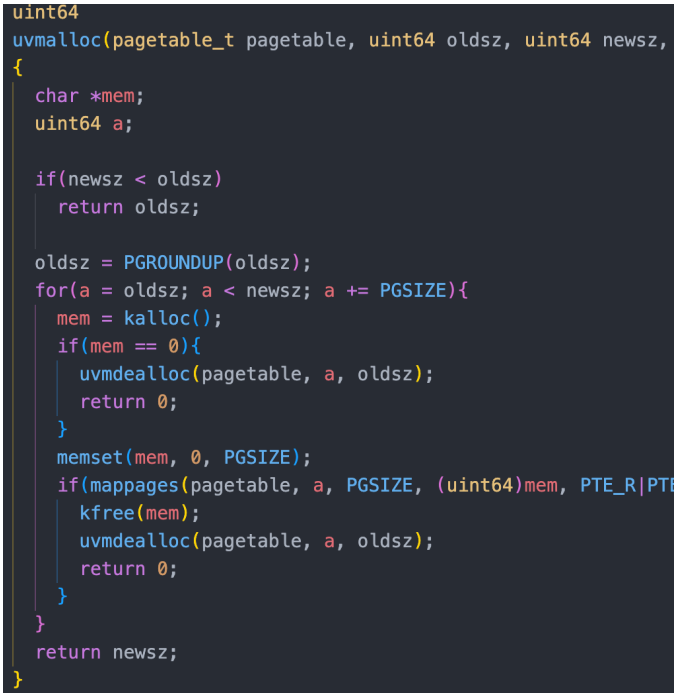
```
void
uvmfirst(pagetable_t pagetable, uchar *src, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("uvmfirst: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pagetable, 0, PGSIZE, (uint64)mem, PTE_W|PTE_R|PT
  memmove(mem, src, sz);
}
```

use kalloc and mappages(va = 0, phy-addr)

and move src data to it.

uvmalloc

extend the space of one user process form oldsz to newsz

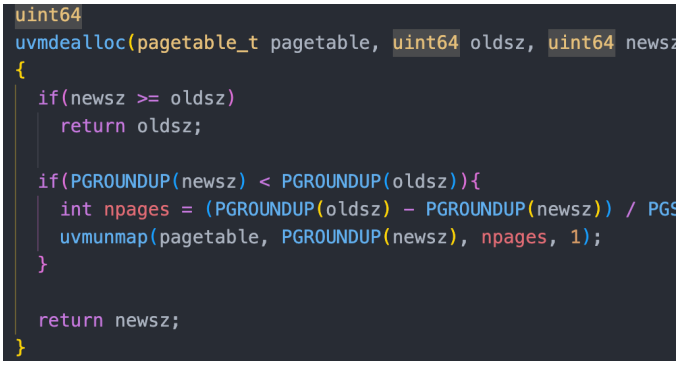


```
uint64
uvmalloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz,
{
  char *mem;
  uint64 a;

  if(newsz < oldsz)
    return oldsz;

  oldsz = PGROUNDUP(oldsz);
  for(a = oldsz; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pagetable, a, PGSIZE, (uint64)mem, PTE_R|PTE
      kfree(mem);
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
  }
  return newsz;
}
```

1. can’t shrink. return old size.
2. new size can not be aligned.
3. kalloc → if(can’t alloc) delete old space of process.
4. if (can’t map) also delete old space

uvmdealloc

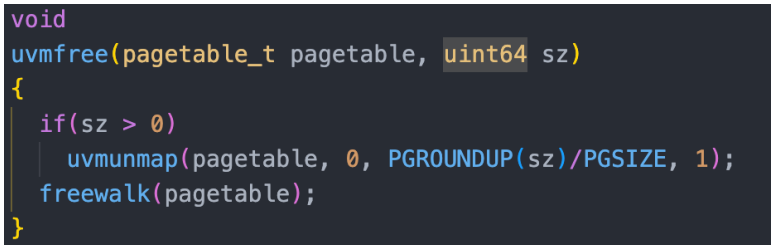
simply call uvmunmap



```
uint64
uvmdealloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz
{
  if(newsz >= oldsz)
    return oldsz;

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

  return newsz;
}
```

uvmfree

delete user memory and free page table



```
void
uvmfree(pagetable_t pagetable, uint64 sz)
{
  if(sz > 0)
    uvmunmap(pagetable, 0, PGROUNDUP(sz)/PGSIZE, 1);
  freewalk(pagetable);
}
```

add one action: freewark()

freewalk

specialized in free page table

```c
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];
    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){
      panic("freewalk: leaf");
    }
  }
  kfree((void*)pagetable);
}
```

first get pagetable, it just the address of root page table.

it also pointer at 512 PTE of middle page table.

if(valid and only this flag was set) it should be a page table, get address of PTE as child and recursively do free walk.

if not only valid(haven't delete the leaf) error

finally ,use kfree to kill address in physical.

uvmcopy

copy parents' memory into child's (both pagetable and memory)

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

  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      panic("uvmcopy: pte should exist");
    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;
}
```

only can copy from address: 0

get flag and kalloc new men → memmove and map the page

if err(can’t alloc or can’t map correctly)

delete all about new.

uvmclear

just clear PTE_user flag

```c
void
uvmclear(pagetable_t pagetable, uint64 va)
{
  pte_t *pte;

  pte = walk(pagetable, va, 0);
  if(pte == 0)
    panic("uvmclear");
  *pte &= ~PTE_U;
}
```

copyout

from kernel to user, given pagetable,

```c
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint6
{
  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;
}
```

we need to use a new function called walkaddr to get physical address

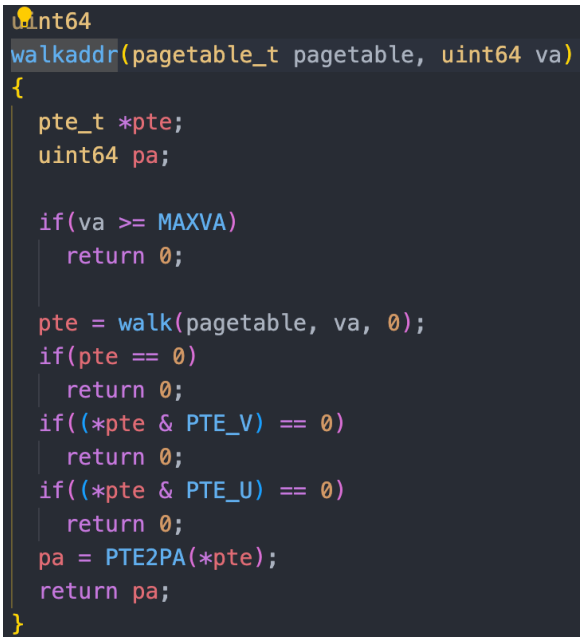
each time , it can only copy for one page(physical)

use memmove to copy

and try to go back to do next page.

walkaddr

just a walk for user mode (parameter could be illegal from user call e.g. copyout…)



```
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;
}
```

it also check valid and user flag

return actual page start address.

copyin

copy from user to kernel

```c
int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint6
{
  uint64 n, va0, pa0;

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

resource address is virtual and dst is physical.

also need to walkaddr() to get physical address and use memmove()

copyinstr

copy a string from user to kernel

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

  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));
    while(n > 0){
      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;
  }
}
```

if got_null or reach max, program will end

it’s action likes copyin but need to notice at ‘\0’

code review about proc(map part)

proc_mapstacks

initialize action

allocate NPROC pages for kernel stack (it’s for each process to have one)

```
void
proc_mapstacks(pagetable_t kpgtbl)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    char *pa = kalloc();
    if(pa == 0)
      panic("kalloc");
    uint64 va = KSTACK((int) (p - proc));
    kvmmap(kpgtbl, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
  }
}
```

use kalloc and kvmmap to new virtual address — va, which is especially made for kstack

```
va = KSTACK(p - proc);
```

proc_pagetable

create a user page table .

but only with trampoline and trapframe.

```
pagetable_t pagetable;

// An empty page table.
pagetable = uvmcreate();
if(pagetable == 0)
  return 0;

// map the trampoline code (for system call return)
// at the highest user virtual address.
// only the supervisor uses it, on the way
// to/from user space, so not PTE_U.
if(mappages(pagetable, TRAMPOLINE, PGSIZE,
            (uint64)trampoline, PTE_R | PTE_X) < 0){
  uvmfree(pagetable, 0);
  return 0;
}

// map the trapframe page just below the trampoline page,
// trampoline.S.
if(mappages(pagetable, TRAPFRAME, PGSIZE,
            (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmfree(pagetable, 0);
  return 0;
}

return pagetable;
```

1. get a page table for user using uvmcreate()
2. map TRAMPOLINE page
3. map TRAPFRAME page

   user memory layout:

   text→original data and bss → stack → heap → syscall → p::trapframe → trampoline(same phypage in kernel)

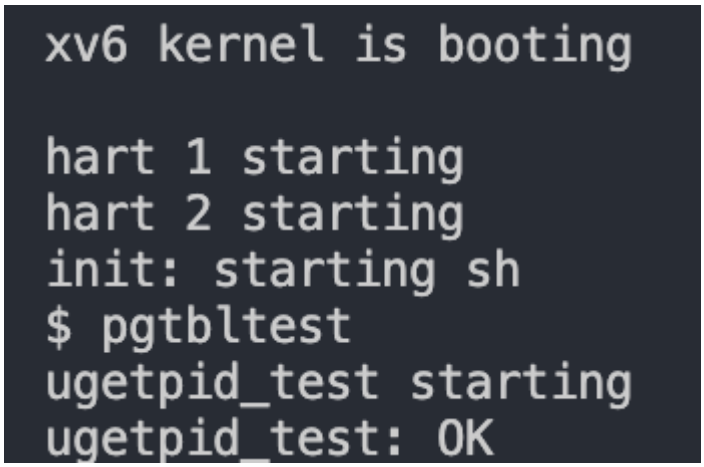
freepagetable

```c
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmfree(pagetable, sz);
}
```

uvmunmap and recursively delete page table

(first two may be too far from addr:0)

lab3-1:speed up syscalls



lab3-2:print a page table

```c
void walk_print(pagetable_t pagetable, int depth)
{

  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if((pte & PTE_V)){
      uint64 child = PTE2PA(pte);
      for(int i=0; i<=depth; i++)
        printf(" ..");
      printf("%d: pte %p pa %p\n", i, pte, child);
      if((pte & (PTE_R|PTE_W|PTE_X)) == 0)
        walk_print((pagetable_t)child, depth + 1);
    }
  }
}

void vmprint(pagetable_t pagetable) {
  printf("page table %p\n", pagetable);
  walk_print(pagetable, 0);
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

hart 2 starting
page table 0x0000000087f6b000
 ..0: pte 0x0000000021fd9c01 pa 0x0000000087f67000
 .. ..0: pte 0x0000000021fd9801 pa 0x0000000087f66000
 .. .. ..0: pte 0x0000000021fda01b pa 0x0000000087f68000
 .. .. ..1: pte 0x0000000021fd9417 pa 0x0000000087f65000
 .. .. ..2: pte 0x0000000021fd9007 pa 0x0000000087f64000
 .. .. ..3: pte 0x0000000021fd8c17 pa 0x0000000087f63000
 ..255: pte 0x0000000021fda801 pa 0x0000000087f6a000
 .. ..511: pte 0x0000000021fda401 pa 0x0000000087f69000
 .. .. ..509: pte 0x0000000021fdcc13 pa 0x0000000087f73000
 .. .. ..510: pte 0x0000000021fdd007 pa 0x0000000087f74000
 .. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$
```

first , we need to print the page table address, so we choose to just print and call walk_print (which is recursive)

then we change the judgement expression, if valid just print, if is a non-leaf PTE, we go next level.

```
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMB
make[1]: Leaving directory '/root/xv6/xv6-labs-2022'
== Test pte printout == pte printout: OK (3.6s)
[xv6-labs-2022]
```

Detect which pages have been accessed

by inspecting the access bits in the RISC-V page table

The RISC-V hardware page walker marks these bits in the PTE whenever it resolves a TLB miss.

implement pgaccess(), a system call that reports which pages have been accessed

three arguments:

First, it takes the starting virtual address of the first user page to check.

Second, it takes the number of pages to check.

Finally, it takes a user address to a buffer to store the results into a bitmask

(a datastructure that uses one bit per page and where the first page corresponds to the least significant bit)

read pgaccess_test()

```c
void
pgaccess_test()
{
  char *buf;
  unsigned int abits;
  printf("pgaccess_test starting\n");
  testname = "pgaccess_test";
  buf = malloc(32 * PGSIZE);
  if (pgaccess(buf, 32, &abits) < 0)
    err("pgaccess failed");
  buf[PGSIZE * 1] += 1;
  buf[PGSIZE * 2] += 1;
  buf[PGSIZE * 30] += 1;
  if (pgaccess(buf, 32, &abits) < 0)
    err("pgaccess failed");
  if (abits != ((1 << 1) | (1 << 2) | (1 << 30)))
    err("incorrect access bits set");
  free(buf);
  printf("pgaccess_test: OK\n");
}
```

allocate 32 page’s space(using malloc)

try to modify some of bytes.

check abits whether correct

read RISCV manual to find PTE_A

Figure 4.14: Sv32 physical address.

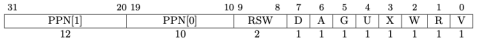


Figure 4.15: Sv32 page table entry.

then we need to find where to set PTE_A

the memory access must go through kernel copy (memmove), so I test the walkaddr,to print some trace information, and successfully get the memory interact with kernel.

finally, I use walk() to find each page (limited 32,because I use the int as bitmap) ,and test PTE_A.

use copyout to transmit the information.