Lab3: page tables

https://pdos.csail.mit.edu/6.828/2021/labs/pgtbl.html

kernel memory layout:

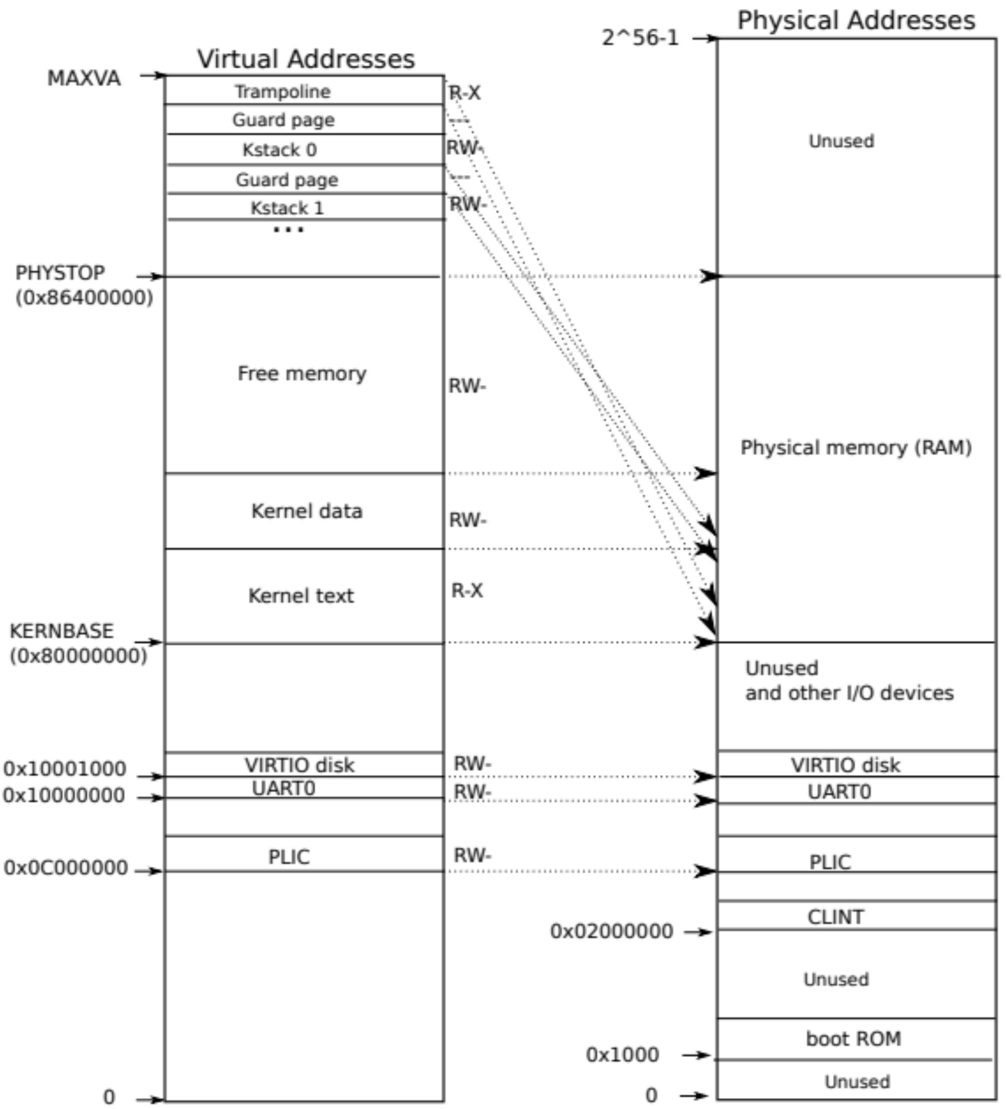

Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

User space process memory layout:

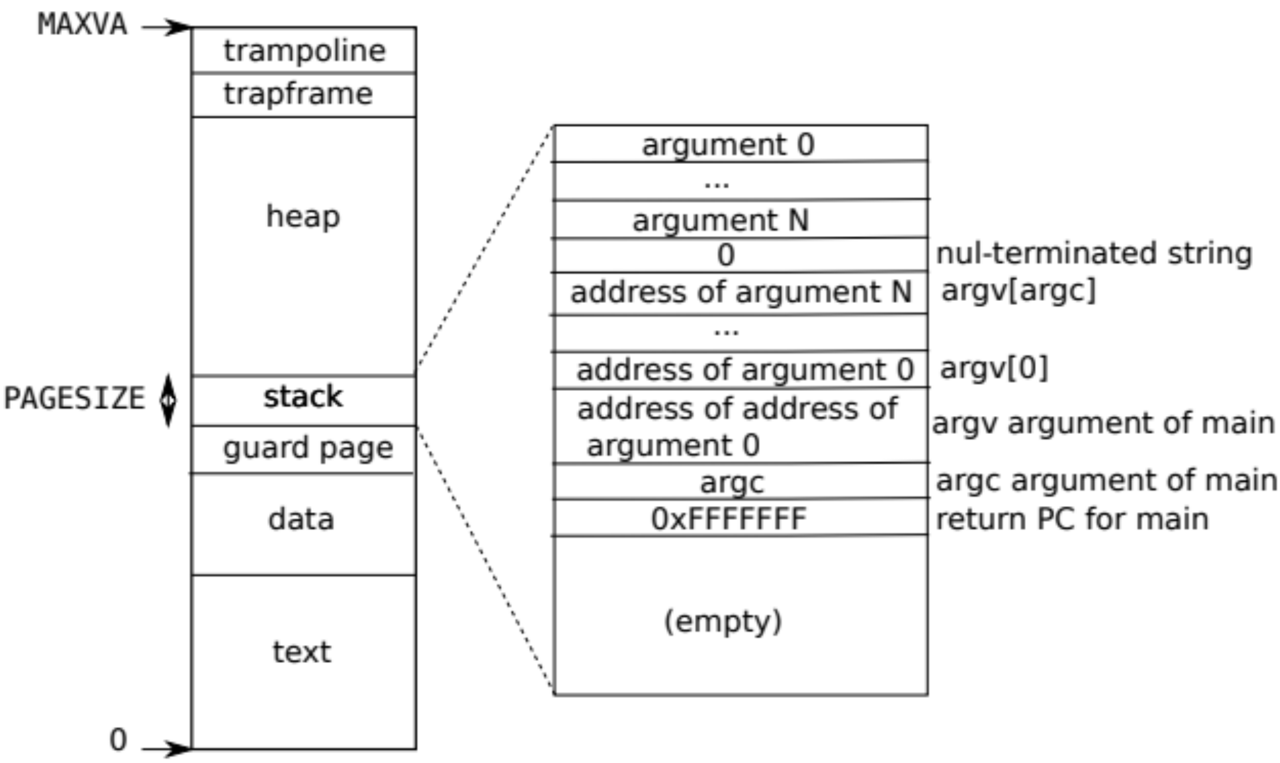

Figure 3.4: A process's user address space, with its initial stack.

function need to use:

`uvmunmap` in vm.c:

```
// Remove npages of mappings starting from va. va must be
// page-aligned. The mappings must exist.
// Optionally free the physical memory.
void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
  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;
  }
}
```

`mappages`

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned. Returns 0 on success, -1 if walk() couldn't
// allocate a needed page-table page.
int
mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
{
  uint64 a, last;
  pte_t *pte;

  if(size == 0)
    panic("mappages: size");

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

note that `va` must not already map to another phy addr.

`walk`

```
// 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)];
}
```

Speed up system calls (easy)

When each process is created, map one read-only page at USYSCALL (a VA defined in `memlayout.h`). At the start of this page, store a `struct usyscall` (also defined in `memlayout.h`), and initialize it to store the PID of the current process.

in `proc_pagetable.c`, add codes to allocate some physical page and map va `USYSCALL` to this.

```
// Create a user page table for a given process,
// with no user memory, but with trampoline pages.
pagetable_t
proc_pagetable(struct proc *p)
{
  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 just below TRAMPOLINE, for 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;
  }
  struct usyscall *usyscall_p;
  if ((usyscall_p = (struct usyscall *)kalloc()) == 0)
  {
    panic("kalloc failed");
  }
  usyscall_p->pid = p->pid;
  if (mappages(pagetable, USYSCALL, PGSIZE,
               (uint64)usyscall_p, PTE_R | PTE_U) < 0)
  {
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmunmap(pagetable, TRAPFRAME, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
  }
  return pagetable;
}
```

Also in function `proc_freepagetable.c` that is called in function `freeproc`, remove the mapping from VA `USYSCALL`.

```
// Free a process's page table, and free the
// physical memory it refers to.
void proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  uvmunmap(pagetable, USYSCALL, 1, 0);
  uvmfree(pagetable, sz);
}
```

result:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgtbltest: pgaccess_test failed: incorrect access bits set, pid=3
```

Print a page table (easy)

add a function in `vm.c` to use DFS to iterate all valid PTEs and print them:

```
void dfs_print(pte_t pte, int level, int index){
  if (level > 2){
    return;
  }
  char *format;
  if (level == 0){
    format = "..%d: pte %p pa %p\n";
  } else if(level == 1){
    format = ".. ..%d: pte %p pa %p\n";
  } else {
    format = ".. .. ..%d: pte %p pa %p\n";
  }

  printf(format, index, pte, PTE2PA(pte));

  pagetable_t pagetable = (pagetable_t)PTE2PA(pte);
  for(int i = 0; i < 512; i ++){
    pte_t pte = pagetable[i];
    if (!(pte & PTE_V)){
      continue;
    }
    dfs_print(pte, level + 1, i);
  }
}

void vmprint(pagetable_t pagetable)
{
  printf("page table %p\n", pagetable);
  for (int i = 0; i < 512; i ++){
    pte_t pte = pagetable[i];
    if (!(pte & PTE_V)) {
      continue;
    }
    dfs_print(pte, 0, i);
  }
```

```
  return;
}
```

also add the definition of the function in `defs.h`:

```
// print page table
void vmprint(pagetable_t pagetable);
```

and the instruction to print pagetable in `exec.c` L#118

```
  if (p->pid == 1)
  {
    vmprint(p->pagetable);
  }
```

Result:

```
== Test pgtbltest ==
$ make qemu-gdb
(4.6s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: FAIL
    ...
         ugetpid_test starting
         ugetpid_test: OK
         pgaccess_test starting
         pgtbltest: pgaccess_test failed: incorrect access bits set, pid=3
         $ qemu-system-riscv64: terminating on signal 15 from pid 698141 (make)
    MISSING '^pgaccess_test: OK$'
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (0.7s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: FAIL
    Cannot read answers-pgtbl.txt
== Test usertests ==
$ make qemu-gdb
```

Detecting which pages have been accessed (hard)

From websearching, the access bit the sixth least significant bit in PTE of RISC-V architecture.

Define `PTE_A` in `risc.h`

```
#define PTE_A (1L << 6) // for lab3
```

Implement the pgaccess syscall in `sysproc.c`:

Basically it use `walk` to retrieve the PTE according to va, and check its access bit.

```
int
sys_pgaccess(void)
{
  // lab pgtbl: your code here.
  // parse argument
  uint64 buf;
  if (argaddr(0, &buf) < 0) {
    return -1;
  }

  int size;
  if (argint(1, &size) < 0) {
    return -1;
  }

  uint64 dstva;
  if (argaddr(2, &dstva) < 0) {
    return -1;
  }

  if (size > 32) {
    printf("pgaccess cannot handle size > 32\n");
  }

  // kernel buffer for storing access bits
  unsigned int abits = 0;

  struct proc *p = myproc();
  uint64 base = PGROUNDDOWN(buf);
  for (int i = 0; i < size; i ++){
    uint64 va = base + PGSIZE * i;
    pte_t *pte = walk(p->pagetable, va, 0);
    if (*pte & PTE_A){
      abits |= (1L << i);
      *pte -= PTE_A;
    }
  }
  if (copyout(p->pagetable, dstva, (char *)&abits, 4) < 0) {
    return -1;
  }
  return 0;
}
```

Result

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
page table 0x0000000087f6e000
..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. ..0: pte 0x0000000021fda001 pa 0x0000000087f68000
.. .. ..0: pte 0x0000000021fda81f pa 0x0000000087f6a000
.. .. ..1: pte 0x0000000021fd9c0f pa 0x0000000087f67000
.. .. ..2: pte 0x0000000021fd981f pa 0x0000000087f66000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..509: pte 0x0000000021fdac13 pa 0x0000000087f6b000
.. .. ..510: pte 0x0000000021fddc07 pa 0x0000000087f77000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
$
```