Lab 3: Page Tables

6.1810 Operating System Engineering

Chapter Review

3.1 Paging Hardware

- Page Tables are used for *virtual address to physical address translation*
- *satp* points to page table
- RISC V xv6: *3 level architecture*
- page table is *4096 bytes,* each row is page table entry (PTE)

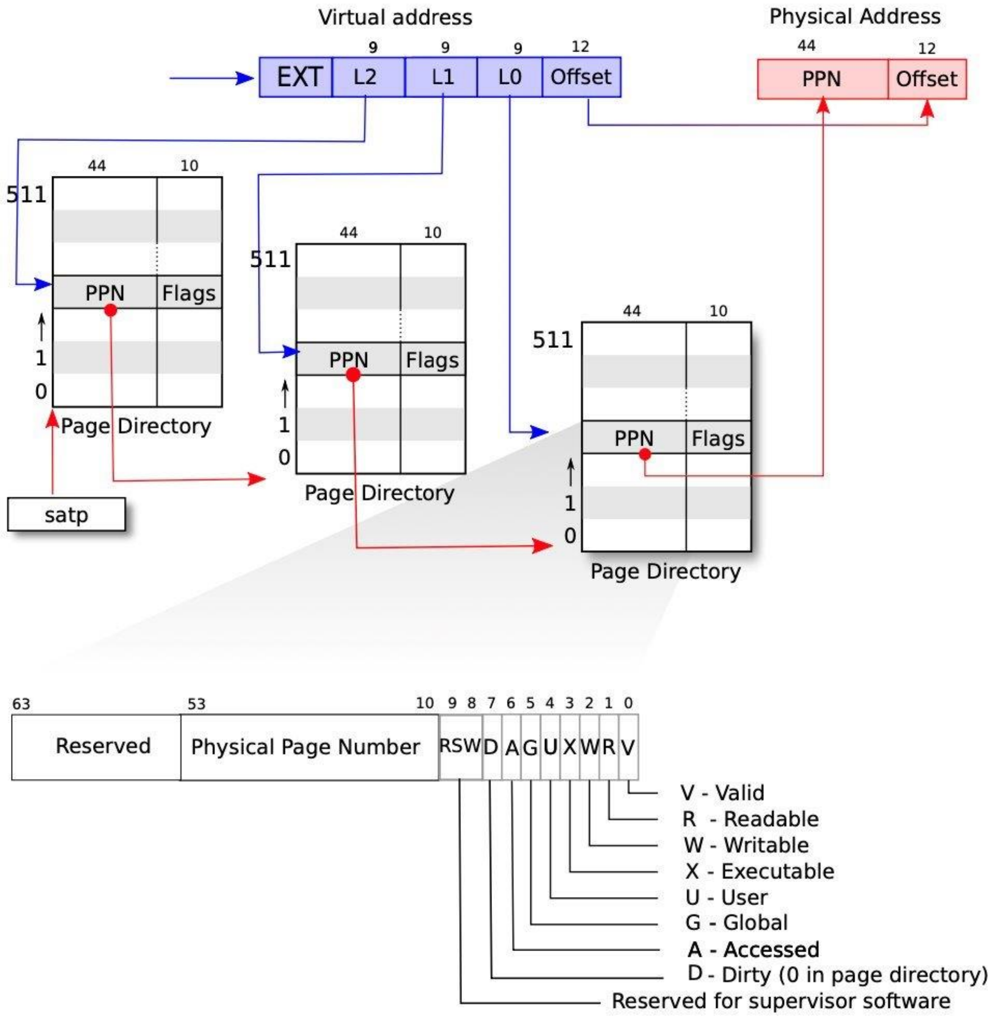
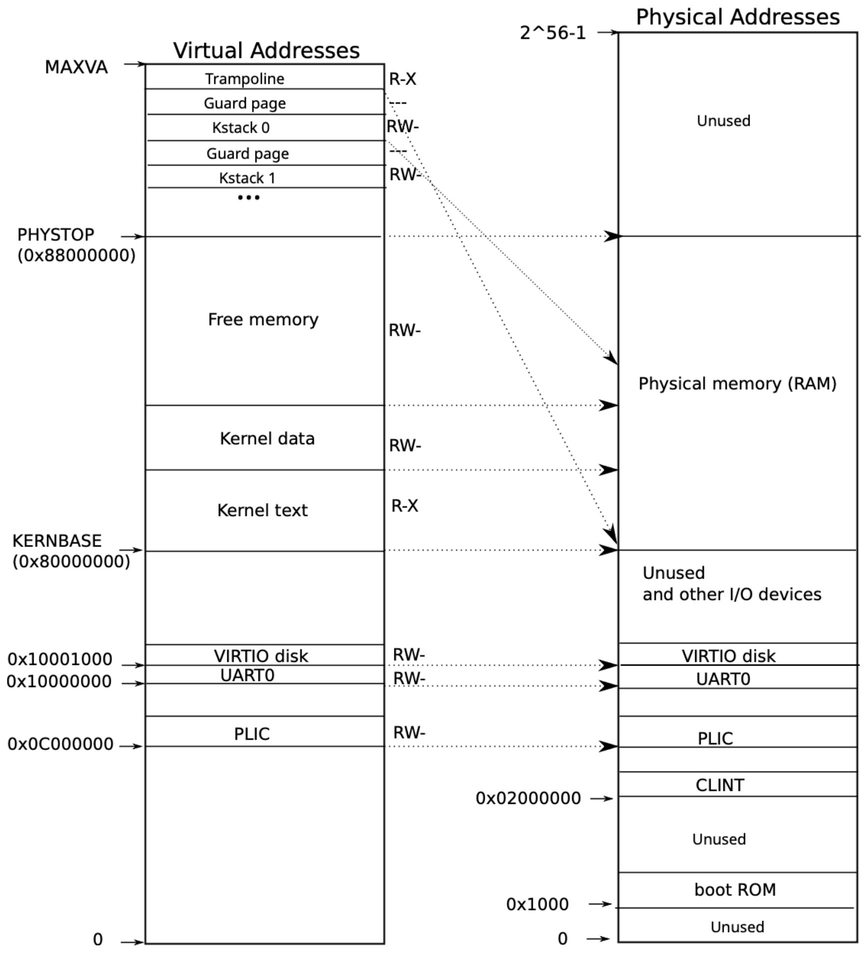


Figure 3.2: RISC-V address translation details.

3.2 Kernel Address Space

- One page table per process
- One shared kernel page table for kernel address space
  - Trampoline page
  - Kernel stack page for preventing overflow



3.3 Address Space

- Conceptually, every load / store / fetch to memory will walk through the whole page table
- For better performance, use *Translation Lookaside Buffers (TLBs)* which *caches* recent PTEs
  - If satp changes, all *TLB cache needs to be flushed*

1. Inspect a page table

Inspect a page table (easy)

Inspect a user-process page table (easy)

To help you understand RISC-V page tables, your first task is to explain the page table for a user process.

Run `make qemu` and run the user program `pgtbltest`. The `print_pgtbl` functions prints out the page-table entries for the first 10 pages and last 10 pages of the `pgtbltest` process as follows:

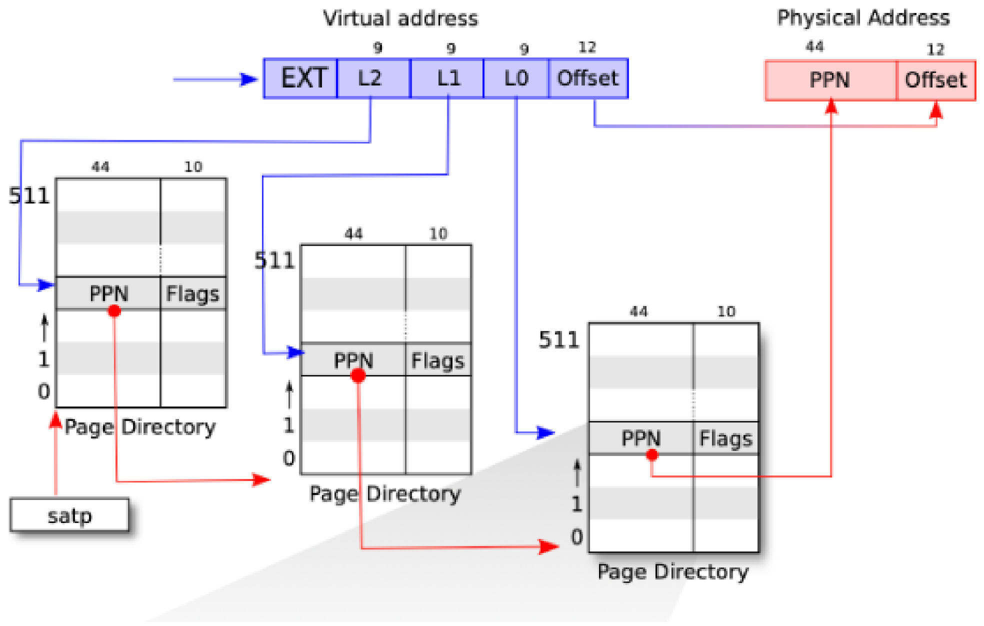
```
va 0 pte 0x21FCF45B pa 0x87F3D000 perm 0x5B
va 1000 pte 0x21FCE85B pa 0x87F3A000 perm 0x5B
...
va 0xFFFFD000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFE000 pte 0x21FD80C7 pa 0x87F60000 perm 0xC7
va 0xFFFFF000 pte 0x20001C4B pa 0x80007000 perm 0x4B
```

- Goal: Let's get familiar with page table concept
- How: By inspecting the actual page table of a sample process given

```
$ pgtbltest
print_pgtbl starting
va 0x0 pte 0x21FC885B pa 0x87F22000 perm 0x5B
va 0x1000 pte 0x21FC7C17 pa 0x87F1F000 perm 0x17
va 0x2000 pte 0x21FC7807 pa 0x87F1E000 perm 0x7
va 0x3000 pte 0x21FC74D7 pa 0x87F1D000 perm 0xD7
va 0x4000 pte 0x0 pa 0x0 perm 0x0
va 0x5000 pte 0x0 pa 0x0 perm 0x0
va 0x6000 pte 0x0 pa 0x0 perm 0x0
va 0x7000 pte 0x0 pa 0x0 perm 0x0
va 0x8000 pte 0x0 pa 0x0 perm 0x0
va 0x9000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFF6000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFF7000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFF8000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFF9000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFA000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFB000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFC000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFD000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFE000 pte 0x21FD08C7 pa 0x87F42000 perm 0xC7
va 0xFFFFF000 pte 0x2000184B pa 0x80006000 perm 0x4B
print_pgtbl: OK
ugetpid_test starting
usertrap(): unexpected scause 0xd pid=4
            sepc=0x57a stval=0x3fffffd000
```

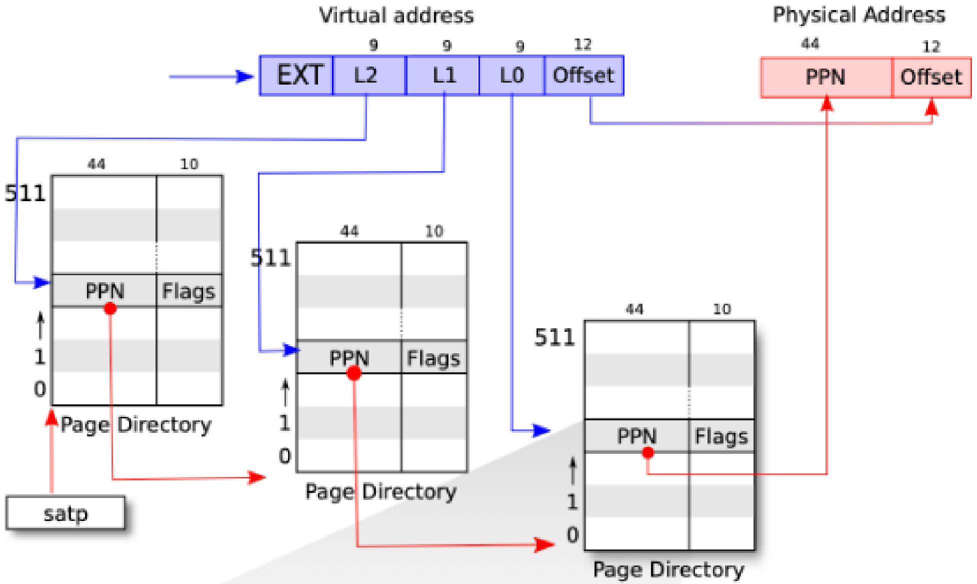
Inspect a page table (easy)

- va: Virtual Address
  - Address that exists for each process or kernel
  - below 39 bits are used for address mapping (from virtual to physical)
  - predefined portion of the virtual address is used as an index to find entry in Page Table



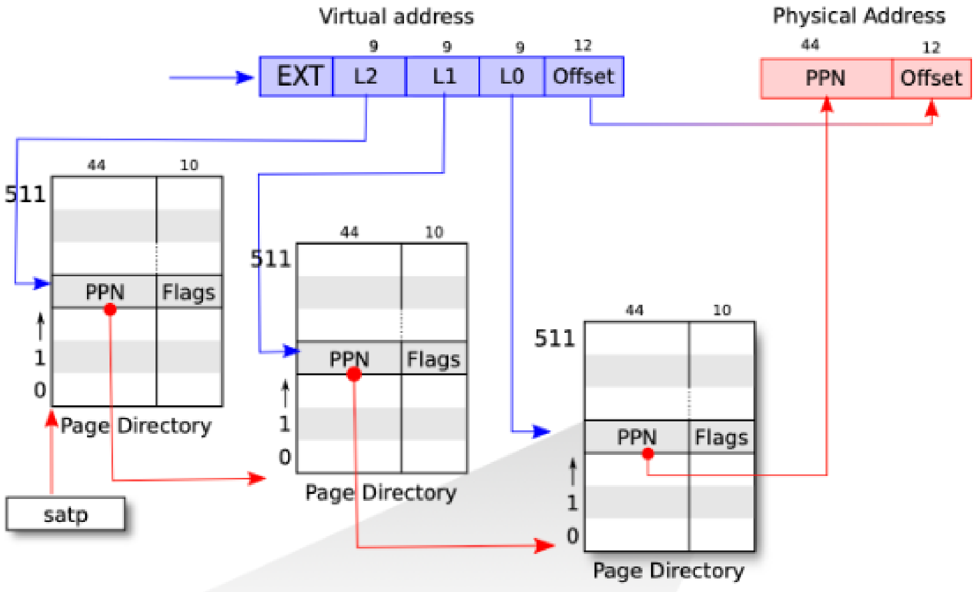
Inspect a page table (easy)

- pte: Page Table Entry
  - Entry that’s identified by a virtual address in Page Table
  - PTE (Page Table Entry) of the lowest level will be used for constructing Physical Address
  - Upper 44 bits are called Physical Page Number (PPN), directly used for Physical Address



Inspect a page table (easy)

- pa: Physical Address
  - An actual memory address
  - Consisted of PPN + Offset
    - PPN: comes from PTE
    - Offset: comes from low 12 bits in Virtual Address



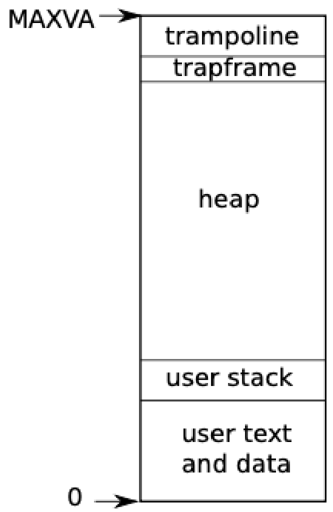
Output & Observation

```
$ pgtbltest
print_pgtbl starting
va 0x0 pte 0x21FC885B pa 0x87F22000 perm 0x5B
va 0x1000 pte 0x21FC7C17 pa 0x87F1F000 perm 0x17
va 0x2000 pte 0x21FC7807 pa 0x87F1E000 perm 0x7
va 0x3000 pte 0x21FC74D7 pa 0x87F1D000 perm 0xD7
va 0x4000 pte 0x0 pa 0x0 perm 0x0
va 0x5000 pte 0x0 pa 0x0 perm 0x0
va 0x6000 pte 0x0 pa 0x0 perm 0x0
va 0x7000 pte 0x0 pa 0x0 perm 0x0
va 0x8000 pte 0x0 pa 0x0 perm 0x0
va 0x9000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFF6000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFF7000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFF8000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFF9000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFA000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFB000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFC000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFD000 pte 0x0 pa 0x0 perm 0x0
va 0xFFFFE000 pte 0x21FD08C7 pa 0x87F42000 perm 0
va 0xFFFFF000 pte 0x2000184B pa 0x80006000 perm 0
print_pgtbl: OK
ugetpid_test starting
usertrap(): unexpected scause 0xd pid=4
            sepc=0x57a stval=0x3fffffd000
```

va	perm (Hex)	perm (Binary)	Meaning
`0x0`	`0x5B`	`1011011`	Valid, Readable, Writable, User, Global.
`0x1000`	`0x17`	`0010111`	Valid, Readable, Writable, User.
`0x2000`	`0x7`	`0000111`	Valid, Readable, Writable.
`0xFFFFE000`	`0xC7`	`11000111`	Valid, Readable, Writable.
Others	`0x0`		Not valid (No mapping)
...			
`0xFFFFE000` (Trapframe)	`0xC7`	`11000111`	Valid, Readable, Writable.
`0xFFFFF000` (Trampoline)	`0x4B`	`1001011`	Valid, Readable, Executable.

2. Speed up system calls

Speed up system calls (easy)

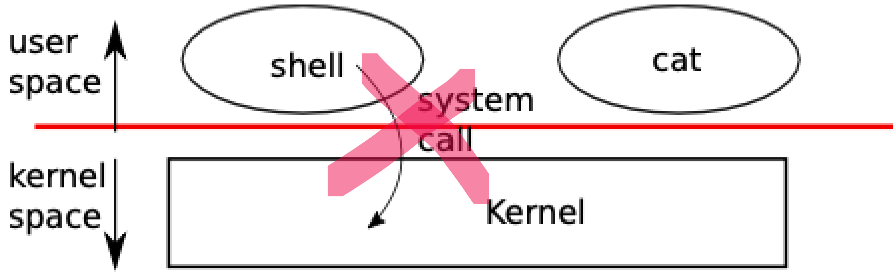


: 2.3: Layout of a process's virtual addres

Some operating systems (e.g., Linux) speed up certain system calls by sharing data in a read-only region between userspace and the kernel. This eliminates the need for kernel crossings when performing these system calls. To help you learn how to insert mappings into a page table, your first task is to implement this optimization for the `getpid()` system call in xv6.

> When each process is created, map one read-only page at USYSCALL (a virtual address 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. For this lab, `ugetpid()` has been provided on the userspace side and will automatically use the USYSCALL mapping. You will receive full credit for this part of the lab if the `ugetpid` test case passes when running `pgtbltest`.

- Goal: Optimizing getpid() system call (== Make it not a system call)
- How: Make the user can access pid without having to do the kernel crossings



Before we start

- HINT: Some data structures we should use are provided

```c
#define USYSCALL (TRAPFRAME - PGSIZ

struct usyscall {
  int pid;  // Process ID
};
```

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

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  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

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // 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 usyscall  *usyscall;  // Shared page
  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)
};
```

mappages

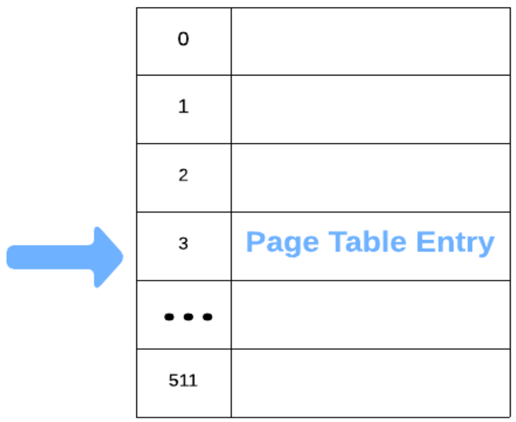
Create Page Table Entries for virtual addresses referring to the given physical addresses; Returns 0 on successs

Using the va as the start point of virtual address,
map virtual address to physical address, page by page.

Use *walk* to get the address of the corresponding address of PTE for current virtual address

Insert valid PTE value with given physical address and permission

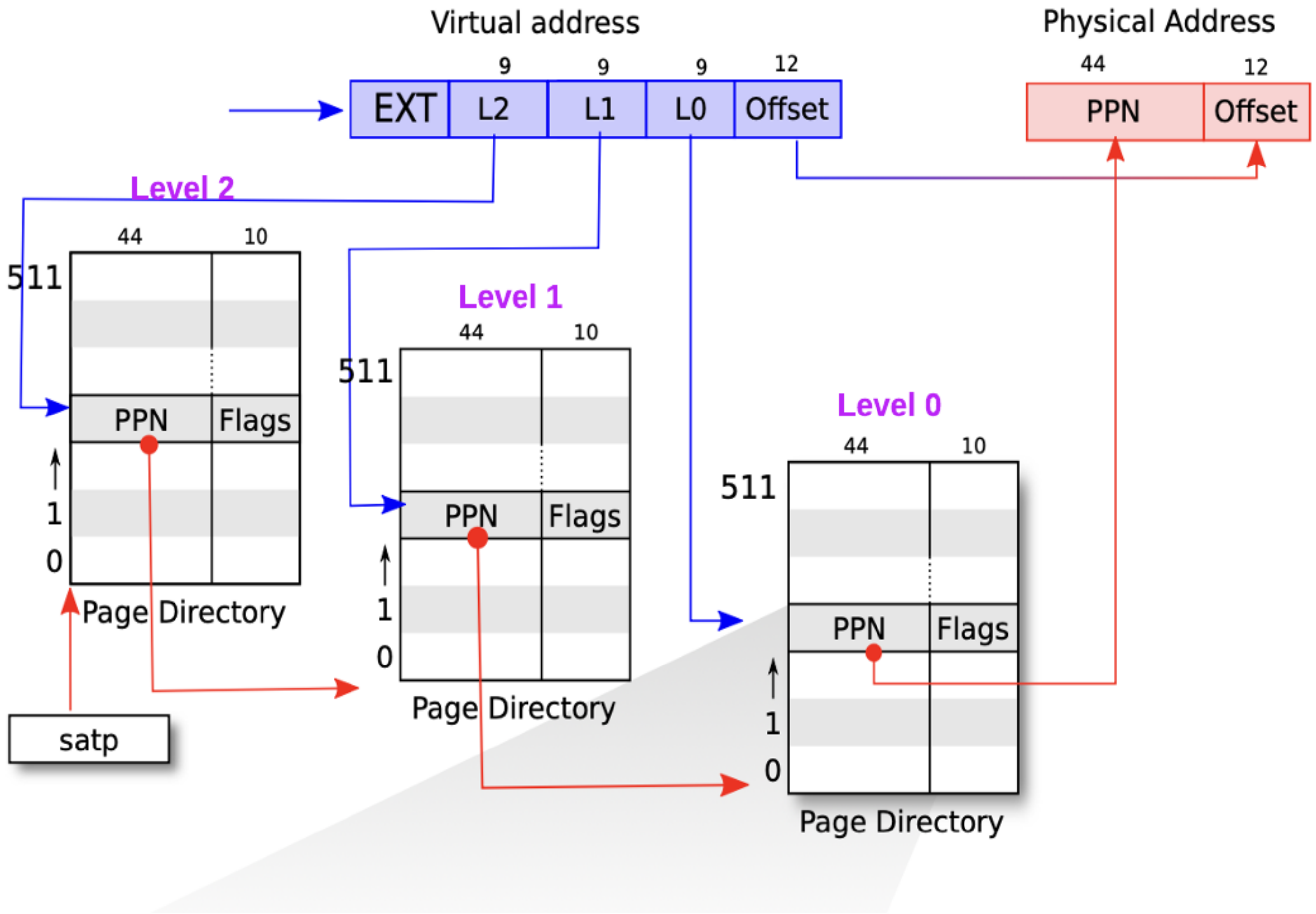
Page Table




```c
int
mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
{
  uint64 a, last;
  pte_t *pte;

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

  if((size % PGSIZE) != 0)
    panic("mappages: size not aligned");

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

  a = va;
  last = va + size - PGSIZE;
  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;
}
```

walk

Return the address of the PTE in page table



Baisically loops through Page Table in Level2, Level1, creating valid PTE if there weren't any.

When valid iteration is done, it will return the lowest Page Table's PTE address (&pagetable[PX(0, 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);
#ifdef LAB_PGTBL
      if(PTE_LEAF(*pte)) {
        return pte;
      }
#endif
    } else {
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
```

allocproc

allocate a memory for a new process; now we add a new field: usyscall for a shared page

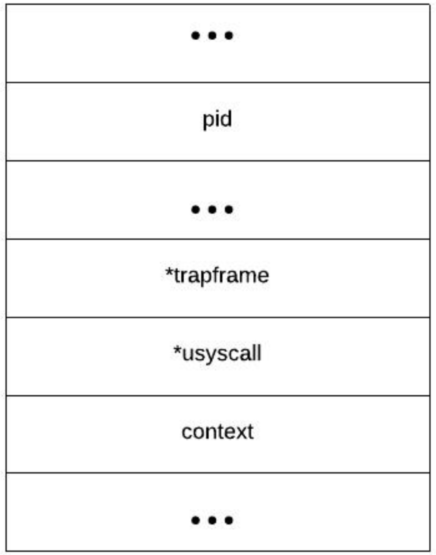
```c
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleepin
  int killed;                  // If non-zero, have be
  int xstate;                  // Exit status to be re
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need
  uint64 kstack;               // Virtual address of k
  uint64 sz;                   // Size of process memo
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampo
  struct usyscall  *usyscall;  // Shared page
  struct context context;      // swtch() here to run
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugg
};
```

```c
found:
  p->pid = allocpid();
  p->state = USED;

  // Allocate a trapframe page.
  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }

  // Allocate and initialize the shared page
  if ((p->usyscall = (struct usyscall *)kalloc()) == 0) {
      freeproc(p);
      release(&p->lock);
      return 0;
  }

  // An empty user page table.
  p->pagetable = proc_pagetable(p);
  if(p->pagetable == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }

  // Set up new context to start executing at forkret,
  // which returns to user space.
  memset(&p->context, 0, sizeof(p->context));
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;
  // Store the PID in the shared page
  p->usyscall->pid = p->pid;

  return p;
}
```

Necessary memory for each fields are allocated using kalloc here.
ex. p->trapframe, p->usyscall

After trapframe & usyscall memories are allocated,
an empty pagetable is created with *proc_pagetable()*

struct proc *p



proc_pagetable

create a user page table for a given process; with trampoline, trapframe, and usyscall pages

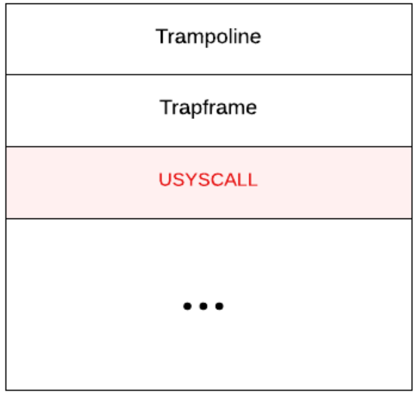
```c
// Create a user page table for a given process, with no user memory,
// but with trampoline and trapframe pages.
pagetable_t
proc_pagetable(struct proc *p)
{
  pagetable_t pagetable;

  // 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 page just below the trampoline page, 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;
  }

  // map the shared page
  if (mappages(pagetable, USYSCALL, PGSIZE,
                (uint64)(p->usyscall), PTE_R | PTE_U) < 0) { // Read-only, User accessible
      uvmunmap(pagetable, TRAPFRAME, 1, 0);
      uvmunmap(pagetable, TRAMPOLINE, 1, 0);
      uvmfree(pagetable, 0);
      return 0;
  }

  return pagetable;
}
```

```c
#define TRAMPOLINE (MAXVA - PGSIZE)
#define TRAPFRAME (TRAMPOLINE - PGSIZE)
#define USYSCALL (TRAPFRAME - PGSIZE)
```

Initialize a new pagetable object with *uvmcreate()*

Map kernel managed pages (physical addresses) to user virtual address with *mappages()*

Virtual Address Space

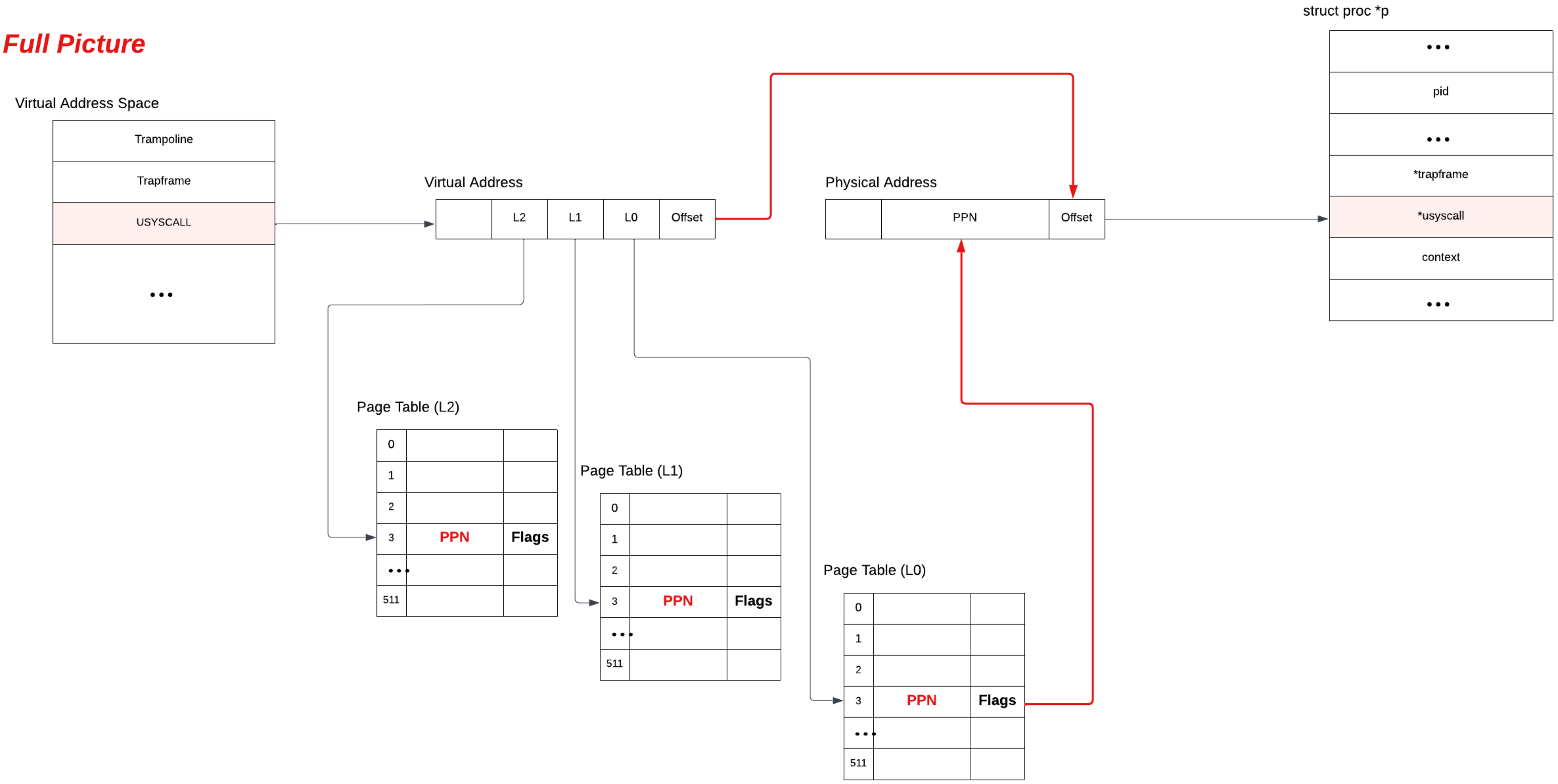


Don't forge to release the resource!

```c
// free a proc structure and the data hanging from it,
// including user pages.
// p->lock must be held.
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;

  // Free the shared page
  if (p->usyscall) {
      kfree((void *)p->usyscall);
  }
  p->usyscall = 0;

  if(p->pagetable)
    proc_freepagetable(p->pagetable, p->sz);
  p->pagetable = 0;
  p->sz = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
}
```

```c
// physical memory it refers to.
void
proc_freepagetable(pagetable_t pagetable, uint64 sz)
{
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmunmap(pagetable, TRAPFRAME, 1, 0);
  // Remove the shared page
  uvmunmap(pagetable, USYSCALL, 1, 0);
  uvmfree(pagetable, sz);
}
```

Full Picture



3. Print a page table

write a function that prints the contents of a page table

```
page table 0x0000000087f22000
 ..0x0000000000000000: pte 0x0000000021fc7801 pa 0x0000000087f1e000
 .. ..0x0000000000000000: pte 0x0000000021fc7401 pa 0x0000000087f1d000
 .. .. ..0x0000000000000000: pte 0x0000000021fc7c5b pa 0x0000000087f1f000
 .. .. ..0x0000000000001000: pte 0x0000000021fc70d7 pa 0x0000000087f1c000
 .. .. ..0x0000000000002000: pte 0x0000000021fc6c07 pa 0x0000000087f1b000
 .. .. ..0x0000000000003000: pte 0x0000000021fc68d7 pa 0x0000000087f1a000
 ..0xffffffffc0000000: pte 0x0000000021fc8401 pa 0x0000000087f21000
 .. ..0xffffffffffe00000: pte 0x0000000021fc8001 pa 0x0000000087f20000
 .. .. ..0xffffffffffffd000: pte 0x0000000021fd4c13 pa 0x0000000087f53000
 .. .. ..0xffffffffffffe000: pte 0x0000000021fd00c7 pa 0x0000000087f40000
 .. .. ..0xfffffffffffff000: pte 0x000000002000184b pa 0x0000000080006000
```

- Print valid PTE row, indicating its depth
- Use Macros
- %p in your printf calls to print out full 64-bit hex PTEs and addresses
- DFS

```c
void
vmprint_pte(pagetable_t pagetable, int depth, uint64 parent_va) {
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if (pte & PTE_V) { // if page table entry is valid
      uint64 va = (i << PXSHIFT(depth)) + parent_va;
      uint64 pa = PTE2PA(pte);
      for (int j = 0; j < 3 - depth; j++)
        printf(" ..");

      // print pte
      printf("%p: pte %p pa %p\n", (void *)va, (void *)pte, (void *)pa);

      // recurse to lower level page table
      if (depth > 0 && !(pte & PTE_LEAF(pte))) {
        vmprint_pte((pagetable_t)pa, depth - 1, va);
      }
    }
  }
}

void
vmprint(pagetable_t pagetable) {
  // your code here
  printf("page table %p\n", (void *)pagetable);
  vmprint_pte(pagetable, 2, 0);
}
```

```c
#if defined(LAB_MMAP) || defined(LAB_PGTBL)
#define PTE_LEAF(pte) ((pte) & (PTE_R | PTE_W | PTE_X))
#endif

// shift a physical address to the right place for a PTE.
#define PA2PTE(pa) ((((uint64)pa) >> 12) << 10)

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

#define PTE_FLAGS(pte) ((pte) & 0x3FF)

// extract the three 9-bit page table indices from a virtual address.
#define PXMASK          0x1FF // 9 bits
#define PXSHIFT(level)  (PGSHIFT+(9*(level)))
#define PX(level, va) ((((uint64) (va)) >> PXSHIFT(level)) & PXMASK)

// one beyond the highest possible virtual address.
// MAXVA is actually one bit less than the max allowed by
// Sv39, to avoid having to sign-extend virtual addresses
// that have the high bit set.
#define MAXVA (1L << (9 + 9 + 9 + 12 - 1))
```

DFS

For valid row print
(level, virtual address, PTE bits, physical address)
root
.. level2
.. .. level1
.. .. .. level0

```
== Test pgtbltest == (0.8s)
== Test   pgtbltest: print_kpgtbl ==
  pgtbltest: print_kpgtbl: FAIL
    ...
         hart 2 starting
         hart 1 starting
    GOOD page table 0x0000000087f4e000
    GOOD  ..0x0000000000000000: pte 0x0000000021fd2801 pa 0x0000000087f4a000
    GOOD  .. ..0x0000000000000000: pte 0x0000000021fd2401 pa 0x0000000087f49000
    GOOD  .. .. ..0x0000000000000000: pte 0x0000000021fd2c1b pa 0x0000000087f4b000
    GOOD  .. .. ..0x0000000000001000: pte 0x0000000021fd2017 pa 0x0000000087f48000
    GOOD  .. .. ..0x0000000000002000: pte 0x0000000021fd1c07 pa 0x0000000087f47000
    GOOD  .. .. ..0x0000000000003000: pte 0x0000000021fd1817 pa 0x0000000087f46000
    GOOD  ..0xffffffc000000000: pte 0x0000000021fd3401 pa 0x0000000087f4d000
    GOOD  .. ..0xffffffffffe00000: pte 0x0000000021fd3001 pa 0x0000000087f4c000
    GOOD  .. .. ..0xffffffffffffe000: pte 0x0000000021fd5807 pa 0x0000000087f56000
    GOOD  .. .. ..0xfffffffffffff000: pte 0x000000002000180b pa 0x0000000080006000
         init: starting sh
         $ pgbltest
    ...
         print_pgtbl: OK
         ugetpid_test starting
         usertrap(): unexpected scause 0xd pid=4
                     sepc=0x57a stval=0x3ffffd000
         $ qemu-system-riscv64: terminating on signal 15 from pid 57706 (<unknown process>)
    MISSING '^ \.\. \.\. \.\.(0xffffffffffffd000|0x0000003ffffd000)'
```

```
00000001 (-------V)
00000001 (-------V)
00011011 (---UX-RV)
00010111 (---U-WRV)
00000111 (-----WRV)
00010111 (---U-WRV)
00000001 (-------V)
00000001 (-------V)
00000111 (-----WRV)
00001011 (----X-RV)
```

4. Use Superpages

Summary of Problem

- Goal
  - Implement "superpages" in the xv6 kernel
- Requirement
  - When a user program calls "sbrk()" with a memory request of 2MB or more, and the requested memory region is 2MB-aligned, the kernel should allocate a superpage.
- Effect
  - This optimization reduces memory usage in the page table and minimizes TLB cache misses, improving program performance.

Hints Provided

1. Analyze the superpg_test test case in user/pgtbltest.c
2. Start with the sys_sbrk() function in kernel/sysproc.c
   1. Trace the sbrk() call to identify how memory allocation works
3. Add superalloc() and superfree() function in kernel/kalloc.c to manage 2MB memory regions
4. Modify uvmcopy() and uvmunmap() to handle superpages during process fork and exit

sys_sbrk() : called by sbrk() to increase memory size

```c
uint64
sys_sbrk(void)
{
  uint64 addr;
  int n;

  argint(0, &n); // 요청된 메모리 크기
  addr = myproc()->sz; // 현재 프로세스 메모리 크기
  if(growproc(n) < 0) // growproc 호출
    return -1;
  return addr; // 이전 메모리 크기를 반환
}
```