Lab4 page tables

目录：

Print a page table

实验目的

> Define a function called vmprint(). It should take a pagetable_t argument, and print that pagetable in the format described below. Insert if(p->pid==1) vmprint(p->pagetable) in exec.c just before the return argc, to print the first process's page table. You receive full credit for this assignment if you pass the pte printout test of make grade.

本实验需要我们添加一个`vmprint()`的函数。它应当接收一个`pagetable_t`作为参数，并以如下格式打印出传进的页表，用于后面两个实验调试用：

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

实验过程

实验提示：

- 你可以将 `vmprint()` 放在 `kernel/vm.c` 中
- 使用定义在 `kernel/riscv.h` 末尾处的宏
- 函数 `freewalk` 可能会对你有所启发
- 将 `vmprint` 的原型定义在 `kernel/defs.h` 中，这样你就可以在 `exec.c` 中调用它了
- 在你的 `printf` 调用中使用 `%p` 来打印像上面示例中的完成的64比特的十六进制 `PTE` 和地址

实验过程：

首先根据提示4，在 `kernel/defs.h` 中添加 `vmprint` 的原型定义，以让我们可以在 `exec.c` 中调用它

```
// vm.c
void            kvminit(void);
......
uint64          walkaddr(pagetable_t, uint64);
int             copyout(pagetable_t, uint64, char *, uint64);
int             copyin(pagetable_t, char *, uint64, uint64);
int             copyinstr(pagetable_t, char *, uint64, uint64);
void            vmprint(pagetable_t pagetable); // 添加函数声明
```

然后就可以在在 `exec.c` 中的 `return argc` 之前插入 `if(p->pid==1) vmprint(p->pagetable)`.

```
//添加打印函数
  if(p->pid==1)
  {
    vmprint(p->pagetable);
  }
  return argc; // this ends up in a0, the first argument to main(argc, argv)
```

接下来我们根据提示1，需要在 `kernel/vm.c` 添加 `vmprint()` 函数。

首先我们看输出格式，可知我们首先需要打印出 `PTE` 的数值。然后参照 `freewalk` 函数进行递归打印。

```
// kernel/vm.c
void pgtblprint(pagetable_t pagetable, int depth) {
  // 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) { // 如果页表项有效
      // 按格式打印页表项
      printf("..");
      for(int j=0;j<depth;j++) {
        printf(" ..");
      }
      printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte));

      // 如果该节点不是叶节点，递归打印其子节点。
      if((pte & (PTE_R|PTE_W|PTE_X)) == 0){
        // this PTE points to a lower-level page table.
        uint64 child = PTE2PA(pte);
        pgtblprint((pagetable_t)child,depth+1);
      }
```

```
    }
  }
}

void vmprint(pagetable_t pagetable) {
  printf("page table %p\n", pagetable);//打印
  pgtblprint(pagetable, 0);//打印页表
}
```

实验结果

```
xv6 kernel is booting

PHYSTOP is 0x0000000088000000
hart 1 starting
hart 2 starting
page table 0x0000000087f64000
..0: pte 0x0000000021fd8001 pa 0x0000000087f60000
.. ..0: pte 0x0000000021fd7c01 pa 0x0000000087f5f000
.. .. ..0: pte 0x0000000021fd841f pa 0x0000000087f61000
.. .. ..1: pte 0x0000000021fd780f pa 0x0000000087f5e000
.. .. ..2: pte 0x0000000021fd741f pa 0x0000000087f5d000
..255: pte 0x0000000021fd8c01 pa 0x0000000087f63000
.. ..511: pte 0x0000000021fd8801 pa 0x0000000087f62000
.. .. ..510: pte 0x0000000021fed807 pa 0x0000000087fb6000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
page table 0x0000000087f0f000
..0: pte 0x0000000021fc2c01 pa 0x0000000087f0b000
.. ..0: pte 0x0000000021fc2801 pa 0x0000000087f0a000
.. .. ..0: pte 0x0000000021fc301f pa 0x0000000087f0c000
.. .. ..1: pte 0x0000000021fc241f pa 0x0000000087f09000
.. .. ..2: pte 0x0000000021fc200f pa 0x0000000087f08000
.. .. ..3: pte 0x0000000021fc1c1f pa 0x0000000087f07000
..255: pte 0x0000000021fc3801 pa 0x0000000087f0e000
.. ..511: pte 0x0000000021fc3401 pa 0x0000000087f0d000
.. .. ..510: pte 0x0000000021fd9407 pa 0x0000000087f65000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

```
hart 1 starting
hart 2 starting
page table 0x0000000087f64000
..0: pte 0x0000000021fd8001 pa 0x0000000087f60000
.. ..0: pte 0x0000000021fd7c01 pa 0x0000000087f5f000
.. .. ..0: pte 0x0000000021fd841f pa 0x0000000087f61000
.. .. ..1: pte 0x0000000021fd780f pa 0x0000000087f5e000
.. .. ..2: pte 0x0000000021fd741f pa 0x0000000087f5d000
..255: pte 0x0000000021fd8c01 pa 0x0000000087f63000
.. ..511: pte 0x0000000021fd8801 pa 0x0000000087f62000
.. .. ..510: pte 0x0000000021fed807 pa 0x0000000087fb6000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
page table 0x0000000087f0f000
..0: pte 0x0000000021fc2c01 pa 0x0000000087f0b000
.. ..0: pte 0x0000000021fc2801 pa 0x0000000087f0a000
.. .. ..0: pte 0x0000000021fc301f pa 0x0000000087f0c000
.. .. ..1: pte 0x0000000021fc241f pa 0x0000000087f09000
.. .. ..2: pte 0x0000000021fc200f pa 0x0000000087f08000
.. .. ..3: pte 0x0000000021fc1c1f pa 0x0000000087f07000
..255: pte 0x0000000021fc3801 pa 0x0000000087f0e000
.. ..511: pte 0x0000000021fc3401 pa 0x0000000087f0d000
.. .. ..510: pte 0x0000000021fd9407 pa 0x0000000087f65000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
$
```

```
tarena@tedu:~/my-xv6-labs-2020$ ./grade-lab-pgtbl pte printout
make: “kernel/kernel”已是最新。
== Test pte printout == pte printout: OK (1.5s)
```


```
tarena@tedu:~/my-xv6-labs-2020$ ./grade-lab-pgtbl pte printout
make: “kernel/kernel”已是最新。
== Test pte printout == pte printout: OK (1.5s)
```


A kernel page table per process

实验目的

> Your first job is to modify the kernel so that every process uses its own copy of the kernel page table when executing in the kernel. Modify struct proc to maintain a kernel page table for each process, and modify the scheduler to switch kernel page tables when switching processes. For this step, each per-process kernel page table should be identical to the existing global kernel page table. You pass this part of the lab if usertests runs correctly.

通过阅读`xv6实验指南`可知Xv6原本的设计是，用户进程在用户态使用各自的用户态页表，但是一旦进入内核态（例如使用了系统调用），则切换到内核页表（通过修改`satp`寄存器，`trampoline.S`）。然而这个内核页表是全局共享的，也就是全部进程进入内核态都共用同一个内核态页表.本实验主要是让每个进程都有自己的内核页表，这样在内核中执行时使用它自己的内核页表的副本。

实验过程

实验提示

- 在`struct proc`中为进程的内核页表增加一个字段
- 为一个新进程生成一个内核页表的合理方案是实现一个修改版的`kvminit`，这个版本中应当创造一个新的页表而不是修改`kernel_pagetable`。你将会考虑在`allocproc`中调用这个函数
- 确保每一个进程的内核页表都关于该进程的内核栈有一个映射。在未修改的`xv6`中，所有的内核栈都在`procinit`中设置。你将要把这个功能部分或全部的迁移到`allocproc`中
- 修改`scheduler()`来加载进程的内核页表到核心的`satp`寄存器(参阅`kvminithart`来获取启发)。不要忘记在调用完`w_satp()`后调用`sfence_vma()`
- 没有进程运行时`scheduler()`应当使用`kernel_pagetable`
- 在`freeproc`中释放一个进程的内核页表
- 你需要一种方法来释放页表，而不必释放叶子物理内存页面。
- 调式页表时，也许`vmprint`能派上用场
- 修改`xv6`本来的函数或新增函数都是允许的；你或许至少需要在`kernel/vm.c`和`kernel/proc.c`中这样做（但不要修改`kernel/vmcopyin.c`，`kernel/stats.c`，`user/usertests.c`，和`user/stats.c`）
- 页表映射丢失很可能导致内核遭遇页面错误。这将导致打印一段包含`sepc=0x00000000XXXXXXXX`的错误提示。你可以在`kernel/kernel.asm`通过查询`XXXXXXXX`来定位错误。

实验过程

首先给根据提示1给在`kernel/proc.h`里面的`struct proc`加上内核页表的字段。

```
//kernel/proc.h
struct proc {
  struct spinlock lock;
   ....
  // 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
  pagetable_t kernelpt;        // 进程的内核页表
  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)
};
```

然后根据提示2，我们需要在`vm.c`中添加新的方法`proc_kpt_init`，该方法用于在`allocproc`中初始化进程的内核页表。这个函数还需要一个辅助函数`uvmmap`，该函数和`kvmmap`方法几乎一致，不同的是`kvmmap`是对Xv6的内核页表进行映射，而`uvmmap`将用于进程的内核页表进行映射。

```
//vm.c
//仿照kvmmap即可
void
uvmmap(pagetable_t pagetable, uint64 va, uint64 pa, uint64 sz, int perm)
{
  if(mappages(pagetable, va, sz, pa, perm) != 0)
    panic("uvmmap");
}

// 为进程创建一个内核页表
```

```
pagetable_t
proc_kpt_init(){
  pagetable_t kernelpt = uvmcreate();
  if (kernelpt == 0) return 0;
  uvmmap(kernelpt, UART0, UART0, PGSIZE, PTE_R | PTE_W);
  uvmmap(kernelpt, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);
  uvmmap(kernelpt, CLINT, CLINT, 0x10000, PTE_R | PTE_W);
  uvmmap(kernelpt, PLIC, PLIC, 0x400000, PTE_R | PTE_W);
  uvmmap(kernelpt, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);
  uvmmap(kernelpt, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R |
PTE_W);
  uvmmap(kernelpt, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);
  return kernelpt;
}
```

然后我们就可以在`kernel/proc.c`里面的`allocproc`添加这个函数的调用(仿照用户页表的创建方式)。

```
//kernel/proc.c
static struct proc*
allocproc(void)
{
  ....
found:
  p->pid = allocpid();

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

  // 初始化内核页表
  p->kernelpt = proc_kpt_init();
  if (p->kernelpt == 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;

    return p;
  }
```

然后根据提示3，为了确保每一个进程的内核页表都关于该进程的内核栈有一个映射。我们需要将`procinit`方法中相关的代码迁移到`allocproc`方法中。很明显就是下面这段代码，将其剪切到上述内核页表初始化的代码之后。

```
// Allocate a page for the process's kernel stack.
// Map it high in memory, followed by an invalid
// guard page.
char *pa = kalloc();
if(pa == 0)
  panic("kalloc");
uint64 va = KSTACK((int) (p - proc));
uvmmap(p->kernelpt, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
p->kstack = va;
```

所以最终我们的`allocproc`函数如下

```
static struct proc*
allocproc(void)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == UNUSED) {
      goto found;
    } else {
      release(&p->lock);
    }
  }
  return 0;

found:
  p->pid = allocpid();

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

  // Init the kernal page table
  p->kernelpt = proc_kpt_init();
  if (p->kernelpt == 0)
  {
    freeproc(p);
    release(&p->lock);
    return 0;
  }
```

```
  // 分配一个物理页，作为新进程的内核栈使用
  char *pa = kalloc();
  if (pa == 0)
    panic("kalloc");
  uint64 va = KSTACK((int)(p - proc));
  uvmmap(p->kernelpt, va, (uint64)pa, PGSIZE, PTE_R | PTE_W);
  p->kstack = va;

  // 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;

  return p;
  }
```

按照提示4，我们需要修改`scheduler()`来加载进程的内核页表到`SATP`寄存器。提示里面请求阅读`kvminithart()`。

```
// Switch h/w page table register to the kernel's page table,
// and enable paging.
void
kvminithart()
{
  w_satp(MAKE_SATP(kernel_pagetable));
  sfence_vma();
}
```

`kvminithart`是用于原先的内核页表，我们将进程的内核页表传进去就可以。在*vm.c*里面添加一个新方法`proc_inithart`，仿照`kvminithart`即可完成对`proc_inithart`的实现。

```
//vm.c
// 加载载进程的内核页表到核心的satp寄存器
void
proc_inithart(pagetable_t kpt){
  w_satp(MAKE_SATP(kpt));
  sfence_vma();
}
```

根据提示5，然后在`scheduler()`内调用即可，但在结束的时候，需要切换回原先的`kernel_pagetable`。直接调用调用上面的`kvminithart()`就能把Xv6的内核页表加载回去。

```
//proc.c
void
scheduler(void)
{
  .......
        p->state = RUNNING;
        c->proc = p;
        //载进程的内核页表到核心的satp寄存器
        proc_inithart(p->kernelpt);
```

```
        swtch(&c->context, &p->context);

        // 切换会原来的内核页表
        kvminithart();
        swtch(&c->context, &p->context);
    .......
}
```

根据提示6，在`freeproc`中释放一个进程的内核页表。首先释放页表内的内核栈，调用`uvmunmap`可以解除映射，最后的一个参数（`do_free`）为一的时候，会释放实际内存。

```
//proc.c
static void
freeproc(struct proc *p)
{
  ....
  p->xstate = 0;
  p->state = UNUSED;
  // 释放内核页表，并释放内核栈
  uvmunmap(p->kernelpt, p->kstack, 1, 1);
  p->kstack = 0;
}
```

然后根据提示7，释放进程的内核页表，先在`kernel/proc.c`里面添加一个方法`proc_freekernelpt`。如下，历遍整个内核页表，然后将所有有效的页表项清空为零。如果这个页表项不在最后一层的页表上，需要继续进行递归。

```
void
proc_freekernelpt(pagetable_t kernelpt)
{

  for(int i = 0; i < 512; i++){
    pte_t pte = kernelpt[i];
    if(pte & PTE_V){
      kernelpt[i] = 0;
      if ((pte & (PTE_R|PTE_W|PTE_X)) == 0){
        uint64 child = PTE2PA(pte);
        proc_freekernelpt((pagetable_t)child);
      }
    }
  }
  kfree((void*)kernelpt);
}
```

然后将需要的函数定义添加到`kernel/defs.h`中

```
// vm.c
void            kvminit(void);
pagetable_t     proc_kpt_init(void); // 用于内核页表的初始化
void            kvminithart(void);
void            proc_inithart(pagetable_t); // 将进程的内核页表保存到SATP寄存器
...
```

最后修改`vm.c`中的`kvmpa`，将原先的`kernel_pagetable`改成`myproc()->kernelpt`，使用进程的内核页表。

```
#include "spinlock.h"
#include "proc.h"

uint64
kvmpa(uint64 va)
{
  uint64 off = va % PGSIZE;
  pte_t *pte;
  uint64 pa;
//在此处将kernel_pagetable改成myproc()->kernelpt
  pte = walk(myproc()->kernelpt, va, 0);
  if(pte == 0)
    panic("kvmpa");
  if((*pte & PTE_V) == 0)
    panic("kvmpa");
  pa = PTE2PA(*pte);
  return pa+off;
}
```

实验结果

```
tarena@tedu:~/my-xv6-labs-2020$ ./grade-lab-pgtbl usertests
make: “kernel/kernel”已是最新。
== Test usertests == (219.6s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
```

```
tarena@tedu:~/my-xv6-labs-2020$ ./grade-lab-pgtbl usertests
make: “kernel/kernel”已是最新。
== Test usertests == (219.6s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
```

Simplify `copyin/copyinstr`

实验目的

> Replace the body of copyin in kernel/vm.c with a call to copyin_new (defined in kernel/vmcopyin.c); do the same for copyinstr and copyinstr_new. Add mappings for user addresses to each process's kernel page table so that copyin_new and copyinstr_new work. You pass this assignment if usertests runs correctly and all the make grade tests pass.

这个实验的目标是，在进程的内核态页表中维护一个用户态页表映射的副本，这样使得内核态也可以对用户态传进来的指针（逻辑地址）进行解引用。这样做相比原来 `copyin` 的实现的优势是，原来的 `copyin` 是通过软件模拟访问页表的过程获取物理地址的，而在内核页表内维护映射副本的话，可以利用 `CPU` 的硬件寻址功能进行寻址，效率更高并且可以受快表加速。

要实现这样的效果，我们需要在每一处内核对用户页表进行修改的时候，将同样的修改也同步应用在进程的内核页表上，使得两个页表的程序段（0 到 PLIC 段）地址空间的映射同步。

实验过程

实验提示

- 先用对`copyin_new`的调用替换`copyin()`，确保正常工作后再去修改`copyinstr`
- 在内核更改进程的用户映射的每一处，都以相同的方式更改进程的内核页表。包括`fork()`，`exec()`，和`sbrk()`.
- 不要忘记在`userinit`的内核页表中包含第一个进程的用户页表
- 用户地址的PTE在进程的内核页表中需要什么权限？(在内核模式下，无法访问设置了`PTE_U`的页面)
- 别忘了上面提到的PLIC限制

实验过程

首先添加复制函数。根据提示5可知，我们在内核模式下，无法访问设置了`PTE_U`的页面，所以我们要将其移除。

```
//vm.c
void u2kvmcopy(pagetable_t pagetable, pagetable_t kernelpt, uint64 oldsz, uint64
newsz)
{
  pte_t *pte_from, *pte_to;
  oldsz = PGROUNDUP(oldsz);
  for (uint64 i = oldsz; i < newsz; i += PGSIZE)
  {
    if ((pte_from = walk(pagetable, i, 0)) == 0)
      panic("u2kvmcopy: src pte does not exist");
    if ((pte_to = walk(kernelpt, i, 1)) == 0)
      panic("u2kvmcopy: pte walk failed");
    uint64 pa = PTE2PA(*pte_from);
    // 关闭读写权限，关闭用户权限，否则kernel无法使用
    uint flags = (PTE_FLAGS(*pte_from)) & (~PTE_U);
    //复制
    *pte_to = PA2PTE(pa) | flags;
  }
}
```

然后根据提示2在内核更改进程的用户映射的每一处 （`fork()`，`exec()`，和`sbrk()`），都复制一份到进程的内核页表。

- `exec()`:

```
//exec.c
int
exec(char *path, char **argv){
  ...
  sp = sz;
  stackbase = sp - PGSIZE;

  // 添加复制逻辑
  u2kvmcopy(pagetable, p->kernelpt, 0, sz);

  // Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
  ...
}
```

- `fork()`:

```
//proc.c
int
fork(void){
  ...
  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;
  ...
  // 复制到新进程的内核页表
  u2kvmcopy(np->pagetable, np->kernelpt, 0, np->sz);
  ...
}
```

- `sbrk()`， 在*kernel/sysproc.c*里面找到`sys_sbrk(void)`，可以知道只有`growproc`是负责将用户内存增加或缩小 n 个字节。根据提示4，为了防止用户进程增长到超过`PLIC`的地址，我们需要给它加个限制。

```
//kernel/sysproc.c
int
growproc(int n)
{
  uint sz;
  struct proc *p = myproc();

  sz = p->sz;
  if(n > 0){
    // 加上PLIC限制
    if (PGROUNDUP(sz + n) >= PLIC){
      return -1;
    }
    if((sz = uvmalloc(p->pagetable, sz, sz + n)) == 0) {
      return -1;
```

```
    }
    // 复制一份到内核页表
    u2kvmcopy(p->pagetable, p->kernelpt, sz - n, sz);
  } else if(n < 0){
    sz = uvmdealloc(p->pagetable, sz, sz + n);
  }
  p->sz = sz;
  return 0;
}
```

然后根据提示1替换掉原有的`copyin()`和`copyinstr()`

```
// Copy from user to kernel.
// Copy len bytes to dst from virtual address srcva in a given page table.
// Return 0 on success, -1 on error.
int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
  return copyin_new(pagetable, dst, srcva, len);
}

// Copy a null-terminated string from user to kernel.
// Copy bytes to dst from virtual address srcva in a given page table,
// until a '\0', or max.
// Return 0 on success, -1 on error.
int
copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
{
  return copyinstr_new(pagetable, dst, srcva, max);
}
```

最后添加到`kernel/defs.h`中

```
// vmcopyin.c
int             copyin_new(pagetable_t, char *, uint64, uint64);
int             copyinstr_new(pagetable_t, char *, uint64, uint64);
```

实验结果

```
== Test usertests == (197.8s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
```

```
== Test usertests == (197.8s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
```

最终结果

```
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (5.5s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (1.4s)
== Test usertests ==
$ make qemu-gdb
(201.4s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 66/66
```

```
$ make qemu-gdb
pte printout: OK (5.5s)
== Test answers-pgtbl.txt == answers-pgtbl.txt: OK
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (1.4s)
== Test usertests ==
$ make qemu-gdb
(201.4s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
== Test time ==
time: OK
Score: 66/66
```

QUESTION

> Explain why the third test is necessary in : give values for and for which the first two test fail (i.e., they will not cause to return -1) but for which the third one is true (resulting in returning -1).`srcva + len < srcva``copyin_new()``srcva``len`Explain why the third test is necessary in : give values for and for which the first two test fail (i.e., they will not cause to return -1) but for which the third one is true (resulting in returning -1).`srcva + len < srcva``copyin_new()``srcva``len`

该函数实现的功能为从虚拟地址`srcva`中拷贝`len`长度的`bytes`到`dst`.我们需要防止数据溢出的情况,所以需要有`srcva + len < srcva`这一个判定标准，一个简单的例子为`srcva=0xf`,`len=2^64-10`,因此`srcva + len=5 < srcva=15`且满足`srcva<p->sz`和`len<p->sz`（64位机`process memory size`即`p->sz`,应为2^{64}）。