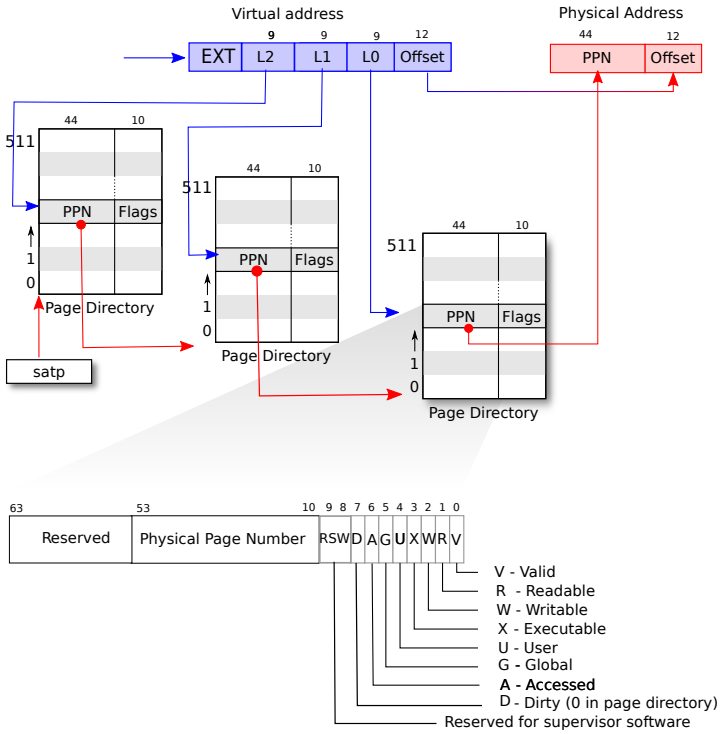

Figure 3.2: RISC-V address translation details.

or write storage. Unlike physical memory and virtual addresses, virtual memory isn't a physical object, but refers to the collection of abstractions and mechanisms the kernel provides to manage physical memory and virtual addresses.

3.2 Kernel address space

Xv6 maintains one page table per process, describing each process's user address space, plus a single page table that describes the kernel's address space. The kernel configures the layout of its address space to give itself access to physical memory and various hardware resources at predictable virtual addresses. Figure 3.3 shows how this layout maps kernel virtual addresses to physical addresses. The file (kernel/memlayout.h) declares the constants for xv6's kernel memory layout.

QEMU simulates a computer that includes RAM (physical memory) starting at physical address `0x80000000` and continuing through at least `0x86400000`, which xv6 calls `PHYSTOP`. The QEMU simulation also includes I/O devices such as a disk interface. QEMU exposes the device interfaces to software as *memory-mapped* control registers that sit below `0x80000000` in the physical address space. The kernel can interact with the devices by reading/writing these special physical addresses; such reads and writes communicate with the device hardware rather than with RAM. Chapter 4 explains how xv6 interacts with devices.

The kernel gets at RAM and memory-mapped device registers using "direct mapping;" that is, mapping the resources at virtual addresses that are equal to the physical address. For example,

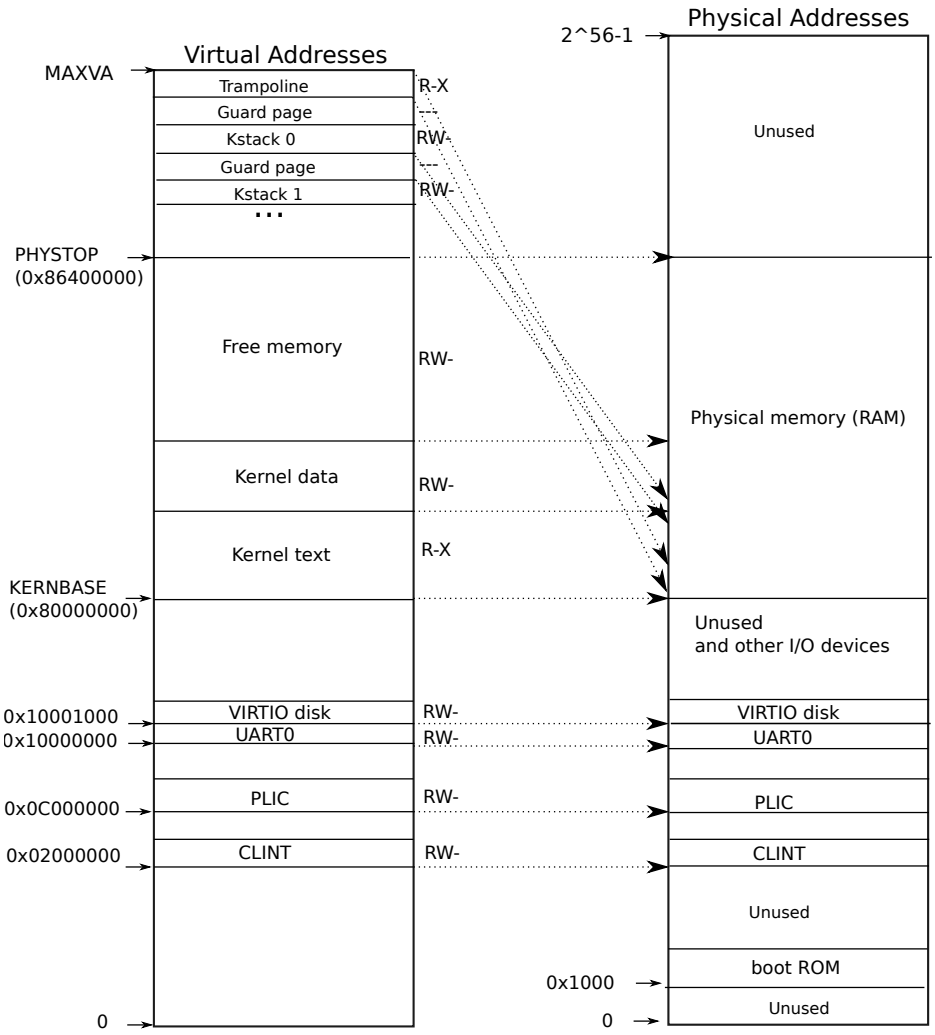

Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

the kernel itself is located at `KERNBASE=0x80000000` in both the virtual address space and in physical memory. Direct mapping simplifies kernel code that reads or writes physical memory. For example, when `fork` allocates user memory for the child process, the allocator returns the physical address of that memory; `fork` uses that address directly as a virtual address when it is copying the parent's user memory to the child.

There are a couple of kernel virtual addresses that aren't direct-mapped:

- The trampoline page. It is mapped at the top of the virtual address space; user page tables have this same mapping. Chapter 4 discusses the role of the trampoline page, but we see here an interesting use case of page tables; a physical page (holding the trampoline code) is mapped twice in the virtual address space of the kernel: once at top of the virtual address space and once with a direct mapping.

- The kernel stack pages. Each process has its own kernel stack, which is mapped high so that below it xv6 can leave an unmapped *guard page*. The guard page's PTE is invalid (i.e., `PTE_V` is not set), so that if the kernel overflows a kernel stack, it will likely cause an exception and the kernel will panic. Without a guard page an overflowing stack would overwrite other kernel memory, resulting in incorrect operation. A panic crash is preferable.

While the kernel uses its stacks via the high-memory mappings, they are also accessible to the kernel through a direct-mapped address. An alternate design might have just the direct mapping, and use the stacks at the direct-mapped address. In that arrangement, however, providing guard pages would involve unmapping virtual addresses that would otherwise refer to physical memory, which would then be hard to use.

The kernel maps the pages for the trampoline and the kernel text with the permissions `PTE_R` and `PTE_X`. The kernel reads and executes instructions from these pages. The kernel maps the other pages with the permissions `PTE_R` and `PTE_W`, so that it can read and write the memory in those pages. The mappings for the guard pages are invalid.

3.3 Code: creating an address space

Most of the xv6 code for manipulating address spaces and page tables resides in `vm.c` (kernel/vm.c:1). The central data structure is `pagetable_t`, which is really a pointer to a RISC-V root page-table page; a `pagetable_t` may be either the kernel page table, or one of the per-process page tables. The central functions are `walk`, which finds the PTE for a virtual address, and `mappages`, which installs PTEs for new mappings. Functions starting with `kvm` manipulate the kernel page table; functions starting with `uvm` manipulate a user page table; other functions are used for both. `copyout` and `copyin` copy data to and from user virtual addresses provided as system call arguments; they are in `vm.c` because they need to explicitly translate those addresses in order to find the corresponding physical memory.

Early in the boot sequence, `main` calls `kvminit` (kernel/vm.c:22) to create the kernel's page table. This call occurs before xv6 has enabled paging on the RISC-V, so addresses refer directly to physical memory. `Kvminit` first allocates a page of physical memory to hold the root page-table page. Then it calls `kvmmap` to install the translations that the kernel needs. The translations include the kernel's instructions and data, physical memory up to `PHYSTOP`, and memory ranges which are actually devices.

`kvmmap` (kernel/vm.c:118) calls `mappages` (kernel/vm.c:149), which installs mappings into a page table for a range of virtual addresses to a corresponding range of physical addresses. It does this separately for each virtual address in the range, at page intervals. For each virtual address to be mapped, `mappages` calls `walk` to find the address of the PTE for that address. It then initializes the PTE to hold the relevant physical page number, the desired permissions (`PTE_W`, `PTE_X`, and/or `PTE_R`), and `PTE_V` to mark the PTE as valid (kernel/vm.c:161).

`walk` (kernel/vm.c:72) mimics the RISC-V paging hardware as it looks up the PTE for a virtual address (see Figure 3.2). `walk` descends the 3-level page table 9 bits at the time. It uses each level's 9 bits of virtual address to find the PTE of either the next-level page table or the final page

(kernel/vm.c:78). If the PTE isn't valid, then the required page hasn't yet been allocated; if the `alloc` argument is set, `walk` allocates a new page-table page and puts its physical address in the PTE. It returns the address of the PTE in the lowest layer in the tree (kernel/vm.c:88).

The above code depends on physical memory being direct-mapped into the kernel virtual address space. For example, as `walk` descends levels of the page table, it pulls the (physical) address of the next-level-down page table from a PTE (kernel/vm.c:80), and then uses that address as a virtual address to fetch the PTE at the next level down (kernel/vm.c:78).

`main` calls `kvminithart` (kernel/vm.c:53) to install the kernel page table. It writes the physical address of the root page-table page into the register `satp`. After this the CPU will translate addresses using the kernel page table. Since the kernel uses an identity mapping, the now virtual address of the next instruction will map to the right physical memory address.

`procinit` (kernel/proc.c:26), which is called from `main`, allocates a kernel stack for each process. It maps each stack at the virtual address generated by `KSTACK`, which leaves room for the invalid stack-guard pages. `kvmmap` adds the mapping PTEs to the kernel page table, and the call to `kvminithart` reloads the kernel page table into `satp` so that the hardware knows about the new PTEs.

Each RISC-V CPU caches page table entries in a *Translation Look-aside Buffer (TLB)*, and when xv6 changes a page table, it must tell the CPU to invalidate corresponding cached TLB entries. If it didn't, then at some point later the TLB might use an old cached mapping, pointing to a physical page that in the meantime has been allocated to another process, and as a result, a process might be able to scribble on some other process's memory. The RISC-V has an instruction `sfence.vma` that flushes the current CPU's TLB. xv6 executes `sfence.vma` in `kvminithart` after reloading the `satp` register, and in the trampoline code that switches to a user page table before returning to user space (kernel/trampoline.S:79).

3.4 Physical memory allocation

The kernel must allocate and free physical memory at run-time for page tables, user memory, kernel stacks, and pipe buffers.

xv6 uses the physical memory between the end of the kernel and `PHYSTOP` for run-time allocation. It allocates and frees whole 4096-byte pages at a time. It keeps track of which pages are free by threading a linked list through the pages themselves. Allocation consists of removing a page from the linked list; freeing consists of adding the freed page to the list.

3.5 Code: Physical memory allocator

The allocator resides in `kalloc.c` (kernel/kalloc.c:1). The allocator's data structure is a *free list* of physical memory pages that are available for allocation. Each free page's list element is a `struct run` (kernel/kalloc.c:17). Where does the allocator get the memory to hold that data structure? It store each free page's `run` structure in the free page itself, since there's nothing else stored there. The free list is protected by a spin lock (kernel/kalloc.c:21-24). The list and the lock are

wrapped in a struct to make clear that the lock protects the fields in the struct. For now, ignore the lock and the calls to `acquire` and `release`; Chapter 6 will examine locking in detail.

The function `main` calls `kinit` to initialize the allocator (kernel/kalloc.c:27). `kinit` initializes the free list to hold every page between the end of the kernel and `PHYSTOP`. `xv6` ought to determine how much physical memory is available by parsing configuration information provided by the hardware. Instead xv6 assumes that the machine has 128 megabytes of RAM. `kinit` calls `freerange` to add memory to the free list via per-page calls to `kfree`. A PTE can only refer to a physical address that is aligned on a 4096-byte boundary (is a multiple of 4096), so `freerange` uses `PGROUNDUP` to ensure that it frees only aligned physical addresses. The allocator starts with no memory; these calls to `kfree` give it some to manage.

The allocator sometimes treats addresses as integers in order to perform arithmetic on them (e.g., traversing all pages in `freerange`), and sometimes uses addresses as pointers to read and write memory (e.g., manipulating the `run` structure stored in each page); this dual use of addresses is the main reason that the allocator code is full of C type casts. The other reason is that freeing and allocation inherently change the type of the memory.

The function `kfree` (kernel/kalloc.c:47) begins by setting every byte in the memory being freed to the value 1. This will cause code that uses memory after freeing it (uses "dangling references") to read garbage instead of the old valid contents; hopefully that will cause such code to break faster. Then `kfree` prepends the page to the free list: it casts `pa` to a pointer to `struct run`, records the old start of the free list in `r->next`, and sets the free list equal to `r`. `kalloc` removes and returns the first element in the free list.

3.6 Process address space

Each process has a separate page table, and when xv6 switches between processes, it also changes page tables. As Figure 2.3 shows, a process's user memory starts at virtual address zero and can grow up to `MAXVA` (kernel/riscv.h:348), allowing a process to address in principle 256 Gigabytes of memory.

When a process asks xv6 for more user memory, xv6 first uses `kalloc` to allocate physical pages. It then adds PTEs to the process's page table that point to the new physical pages. Xv6 sets the `PTE_W`, `PTE_X`, `PTE_R`, `PTE_U`, and `PTE_V` flags in these PTEs. Most processes do not use the entire user address space; xv6 leaves `PTE_V` clear in unused PTEs.

We see here a few nice examples of use of page tables. First, different processes' page tables translate user addresses to different pages of physical memory, so that each process has private user memory. Second, each process sees its memory as having contiguous virtual addresses starting at zero, while the process's physical memory can be non-contiguous. Third, the kernel maps a page with trampoline code at the top of the user address space, thus a single page of physical memory shows up in all address spaces.

Figure 3.4 shows the layout of the user memory of an executing process in xv6 in more detail. The stack is a single page, and is shown with the initial contents as created by exec. Strings containing the command-line arguments, as well as an array of pointers to them, are at the very top of the stack. Just under that are values that allow a program to start at `main` as if the function

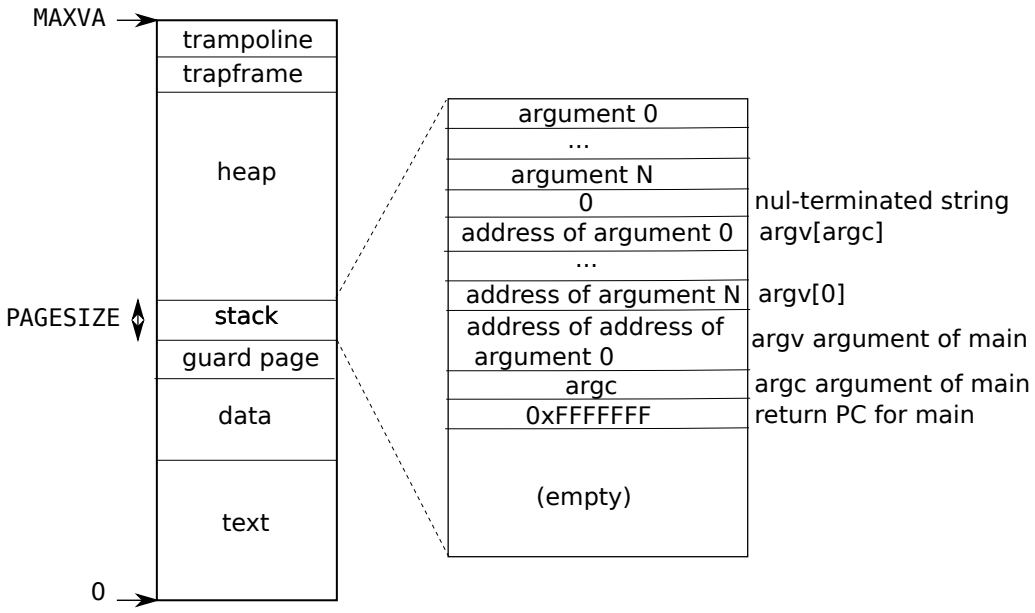

Figure 3.4: A process's user address space, with its initial stack.

`main(argc, argv)` had just been called.

To detect a user stack overflowing the allocated stack memory, xv6 places an invalid guard page right below the stack. If the user stack overflows and the process tries to use an address below the stack, the hardware will generate a page-fault exception because the mapping is not valid. A real-world operating system might instead automatically allocate more memory for the user stack when it overflows.

3.7 Code: sbrk

`Sbrk` is the system call for a process to shrink or grow its memory. The system call is implemented by the function `growproc` (kernel/proc.c:239). `growproc` calls `uvmalloc` or `uvmdealloc`, depending on whether `n` is postive or negative. `uvmalloc` (kernel/vm.c:229) allocates physical memory with `kalloc`, and adds PTEs to the user page table with `mappages`. `uvmdealloc` calls `uvmunmap` (kernel/vm.c:174), which uses `walk` to find PTEs and `kfree` to free the physical memory they refer to.

xv6 uses a process's page table not just to tell the hardware how to map user virtual addresses, but also as the only record of which physical memory pages are allocated to that process. That is the reason why freeing user memory (in `uvmunmap`) requires examination of the user page table.

3.8 Code: exec

`Exec` is the system call that creates the user part of an address space. It initializes the user part of an address space from a file stored in the file system. `Exec` (kernel/exec.c:13) opens the named binary `path` using `namei` (kernel/exec.c:26), which is explained in Chapter 8. Then, it reads the ELF header. Xv6 applications are described in the widely-used *ELF format*, defined in (kernel/elf.h). An ELF binary consists of an ELF header, `struct elfhdr` (kernel/elf.h:6), followed by a sequence of program section headers, `struct proghdr` (kernel/elf.h:25). Each `proghdr` describes a section of the application that must be loaded into memory; xv6 programs have only one program section header, but other systems might have separate sections for instructions and data.

The first step is a quick check that the file probably contains an ELF binary. An ELF binary starts with the four-byte "magic number" `0x7F`, `'E'`, `'L'`, `'F'`, or `ELF_MAGIC` (kernel/elf.h:3). If the ELF header has the right magic number, `exec` assumes that the binary is well-formed.

`Exec` allocates a new page table with no user mappings with `proc_pagetable` (kernel/exec.c:38), allocates memory for each ELF segment with `uvmalloc` (kernel/exec.c:52), and loads each segment into memory with `loadseg` (kernel/exec.c:10). `loadseg` uses `walkaddr` to find the physical address of the allocated memory at which to write each page of the ELF segment, and `readi` to read from the file.

The program section header for `/init`, the first user program created with `exec`, looks like this:

```
# objdump -p _init
user/_init:     file format elf64-littleriscv

Program Header:
    LOAD off    0x00000000000000b0 vaddr 0x0000000000000000
                                   paddr 0x0000000000000000 align 2**3
         filesz 0x0000000000000840 memsz 0x0000000000000858 flags rwx
   STACK off    0x0000000000000000 vaddr 0x0000000000000000
                                   paddr 0x0000000000000000 align 2**4
         filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-
```

The program section header's `filesz` may be less than the `memsz`, indicating that the gap between them should be filled with zeroes (for C global variables) rather than read from the file. For `/init`, `filesz` is 2112 bytes and `memsz` is 2136 bytes, and thus `uvmalloc` allocates enough physical memory to hold 2136 bytes, but reads only 2112 bytes from the file `/init`.

Now `exec` allocates and initializes the user stack. It allocates just one stack page. `Exec` copies the argument strings to the top of the stack one at a time, recording the pointers to them in `ustack`. It places a null pointer at the end of what will be the `argv` list passed to `main`. The first three entries in `ustack` are the fake return program counter, `argc`, and `argv` pointer.

`Exec` places an inaccessible page just below the stack page, so that programs that try to use more than one page will fault. This inaccessible page also allows `exec` to deal with arguments that are too large; in that situation, the `copyout` (kernel/vm.c:355) function that `exec` uses to copy arguments to the stack will notice that the destination page is not accessible, and will return -1.

During the preparation of the new memory image, if `exec` detects an error like an invalid program segment, it jumps to the label `bad`, frees the new image, and returns -1. `Exec` must wait

to free the old image until it is sure that the system call will succeed: if the old image is gone, the system call cannot return -1 to it. The only error cases in `exec` happen during the creation of the image. Once the image is complete, `exec` can commit to the new page table (kernel/exec.c:113) and free the old one (kernel/exec.c:117).

`Exec` loads bytes from the ELF file into memory at addresses specified by the ELF file. Users or processes can place whatever addresses they want into an ELF file. Thus `exec` is risky, because the addresses in the ELF file may refer to the kernel, accidentally or on purpose. The consequences for an unwary kernel could range from a crash to a malicious subversion of the kernel's isolation mechanisms (i.e., a security exploit). xv6 performs a number of checks to avoid these risks. For example `if(ph.vaddr + ph.memsz < ph.vaddr)` checks for whether the sum overflows a 64-bit integer. The danger is that a user could construct an ELF binary with a `ph.vaddr` that points to a user-chosen address, and `ph.memsz` large enough that the sum overflows to 0x1000, which will look like a valid value. In an older version of xv6 in which the user address space also contained the kernel (but not readable/writable in user mode), the user could choose an address that corresponded to kernel memory and would thus copy data from the ELF binary into the kernel. In the RISC-V version of xv6 this cannot happen, because the kernel has its own separate page table; `loadseg` loads into the process's page table, not in the kernel's page table.

It is easy for a kernel developer to omit a crucial check, and real-world kernels have a long history of missing checks whose absence can be exploited by user programs to obtain kernel privileges. It is likely that xv6 doesn't do a complete job of validating user-level data supplied to the kernel, which a malicious user program might be able to exploit to circumvent xv6's isolation.

3.9 Real world

Like most operating systems, xv6 uses the paging hardware for memory protection and mapping. Most operating systems make far more sophisticated use of paging than xv6 by combining paging and page-fault exceptions, which we will discuss in Chapter 4.

Xv6 is simplified by the kernel's use of a direct map between virtual and physical addresses, and by its assumption that there is physical RAM at address 0x8000000, where the kernel expects to be loaded. This works with QEMU, but on real hardware it turns out to be a bad idea; real hardware places RAM and devices at unpredictable physical addresses, so that (for example) there might be no RAM at 0x8000000, where xv6 expect to be able to store the kernel. More serious kernel designs exploit the page table to turn arbitrary hardware physical memory layouts into predictable kernel virtual address layouts.

RISC-V supports protection at the level of physical addresses, but xv6 doesn't use that feature.

On machines with lots of memory it might make sense to use RISC-V's support for "super pages." Small pages make sense when physical memory is small, to allow allocation and page-out to disk with fine granularity. For example, if a program uses only 8 kilobytes of memory, giving it a whole 4-megabyte super-page of physical memory is wasteful. Larger pages make sense on machines with lots of RAM, and may reduce overhead for page-table manipulation.

The xv6 kernel's lack of a `malloc`-like allocator that can provide memory for small objects prevents the kernel from using sophisticated data structures that would require dynamic allocation.

Memory allocation is a perennial hot topic, the basic problems being efficient use of limited memory and preparing for unknown future requests [7]. Today people care more about speed than space efficiency. In addition, a more elaborate kernel would likely allocate many different sizes of small blocks, rather than (as in xv6) just 4096-byte blocks; a real kernel allocator would need to handle small allocations as well as large ones.

3.10 Exercises

1. Parse RISC-V's device tree to find the amount of physical memory the computer has.
2. Write a user program that grows its address space by one byte by calling `sbrk(1)`. Run the program and investigate the page table for the program before the call to `sbrk` and after the call to `sbrk`. How much space has the kernel allocated? What does the PTE for the new memory contain?
3. Modify xv6 to use super pages for the kernel.
4. Modify xv6 so that when a user program dereferences a null pointer, it will receive an exception. That is, modify xv6 so that virtual address 0 isn't mapped for user programs.
5. Unix implementations of `exec` traditionally include special handling for shell scripts. If the file to execute begins with the text `#!`, then the first line is taken to be a program to run to interpret the file. For example, if `exec` is called to run `myprog arg1` and `myprog` 's first line is `#!/interp`, then `exec` runs `/interp` with command line `/interp myprog arg1`. Implement support for this convention in xv6.
6. Implement address space randomization for the kernel.