Prerequisite Knowledge

xv6 Basic Introduction

The operating system, xv6, provides the basic interfaces introduced by Ken Thompson and Dennis Ritchie's Unix operating system, as well as mimicking Unix's internal design. Unix provides a narrow interface whose mechanisms combine well, offering a surprising degree of generality. This interface has been so successful that modern operating systems—BSD, Linux, macOS, Solaris, and even, to a lesser extent, Microsoft Windows—have Unix-like interfaces. Understanding xv6 is a good start toward understanding any of these systems and many others.

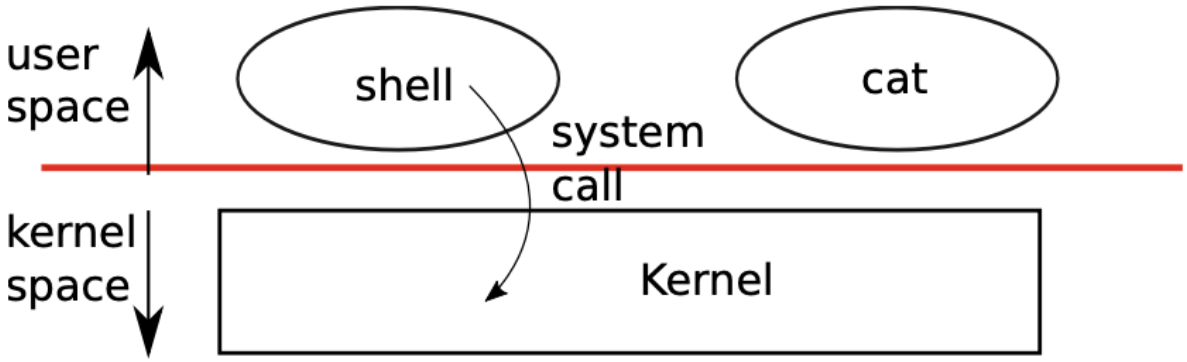

Figure 1.1: A kernel and two user processes.

As Figure 1.1 shows, xv6 takes the traditional form of a ***kernel***, a special program that provides services to running programs. Each running program, called a *process*, has memory containing instructions, data, and a stack. The instructions implement the program's computation. The data are the variables on which the computation acts. The stack organizes the program's procedure calls. A given computer typically has many processes but only a single kernel.

When a process needs to invoke a kernel service, it invokes a ***system call***, one of the calls in the operating system's interface. The system call enters the kernel; the

kernel performs the service and returns. Thus, a process alternates between executing in ***user space*** **and** ***kernel space*****. The k**ernel uses the hardware protection mechanisms provided by a CPU to ensure that **each process executing in user space can access only its own memory.**

The collection of system calls that a kernel provides is the interface that user programs see. The xv6 kernel provides a subset of the services and system calls that Unix kernels traditionally offer. Figure 1.2 lists all of xv6's system calls.

System call	Description
int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int stat(char *file, struct stat *st)	Place info about a named file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

Figure 1.2: Xv6 system calls. If not otherwise stated, these calls return 0 for no error, and -1 if there's an error.

The Unix system call interface has been standardized through the Portable Operating System Interface (POSIX) standard. Xv6 is *not* POSIX compliant: it is missing many system calls (including basic ones such as `lseek`), and many of the system calls it does provide differ from the standard. Our main goals for xv6 are simplicity and clarity while providing a simple UNIX-like system-call interface. Several people have extended xv6 with a few more system calls and a simple C library in order to run basic Unix programs. Modern kernels, however, provide

many more system calls, and many more kinds of kernel services, than xv6. For example, they support networking, windowing systems, user-level threads, drivers for many devices, and so on. Modern kernels evolve continuously and rapidly, and offer many features beyond POSIX.

File Descriptor

A ***file descriptor*** is a small integer representing a kernel-managed object that a process may read from or write to. A process may obtain a file descriptor by opening a file, directory, or device, or by creating a pipe, or by duplicating an existing descriptor. For simplicity, we'll often refer to the object a file descriptor refers to as a "file"; the file descriptor interface abstracts away the differences between files, pipes, and devices, making them all look like streams of bytes.

Internally, the xv6 kernel uses the file descriptor as an index into a per-process table, so that every process has a private space of file descriptors starting at zero.

The `read` and `write` system calls read bytes from and write bytes to open files named by file descriptors. The call `read(fd, buf, n)` reads at most n bytes from the file descriptor `fd`, copies them into `buf`, and returns the number of bytes read. Each file descriptor that refers to a file has an offset associated with it. `read` reads data from the current file offset and then advances that offset by the number of bytes read: a subsequent read will return the bytes following the ones returned by the first `read`. When there are no more bytes to read, `read` returns zero to indicate the end of the file.

The call `write(fd, buf, n)` writes n bytes from `buf` to the file descriptor `fd` and returns the number of bytes written. Fewer than n bytes are written only when an error occurs. Like `read`, `write` writes data at the current file offset and then advances that offset by the number of bytes written: each `write` picks up where the previous one left off.

The `close` system call releases a file descriptor, making it free for reuse by a future `open`, `pipe`, or `dup` system call (see below). A newly allocated file descriptor is always the lowest-numbered unused descriptor of the current process.

The `dup` system call duplicates an existing file descriptor, returning a new one that refers to the same underlying file object. Both file descriptors share an offset, just

as the file descriptors duplicated by `fork` do. This is another way to write `hello world` into a file:

```
fd = dup(1);
write(1, "hello ", 6);
write(fd, "world\n", 6);
```

File

A file's name is distinct from the file itself; the same underlying file, called an ***inode***, can have multiple names, called ***links***. An inode holds ***metadata*** about a file, including its type (file or directory or device), its length, the location of the file's content on disk, and the number of links to a file. It is noticeable that Unix/Linux system does not identify the file's name. Instead, the system uses inode to recognize the file.

Page Tables

Page tables are the most popular mechanism through which the operating system provides each process with its own private address space and memory. Page tables determine what memory addresses mean, and what parts of physical memory can be accessed. They allow xv6 to isolate different process's address spaces and to multiplex them onto a single physical memory.

As a reminder, RISC-V instructions (both user and kernel) manipulate virtual addresses. The ma- chine's RAM, or physical memory, is indexed with physical addresses. The RISC-V page table hardware connects these two kinds of addresses, by mapping each virtual address to a physical address.

Xv6 runs on Sv39 RISC-V, which means that only the bottom 39 bits of a 64-bit virtual address are used; the top 25 bits are not used. In this Sv39 configuration, 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. The paging hardware translates a virtual address by using the top 27 bits of the 39 bits to index into the page table to find a PTE, and making a 56-bit physical address whose top 44 bits come from the PPN in the PTE and whose bottom 12

bits are copied from the original virtual address. A **page table** gives the operating system control over virtual-to physical address translations at the granularity of aligned chunks of 4096 (2^{12}) bytes. Such a chunk is called a ***page***.

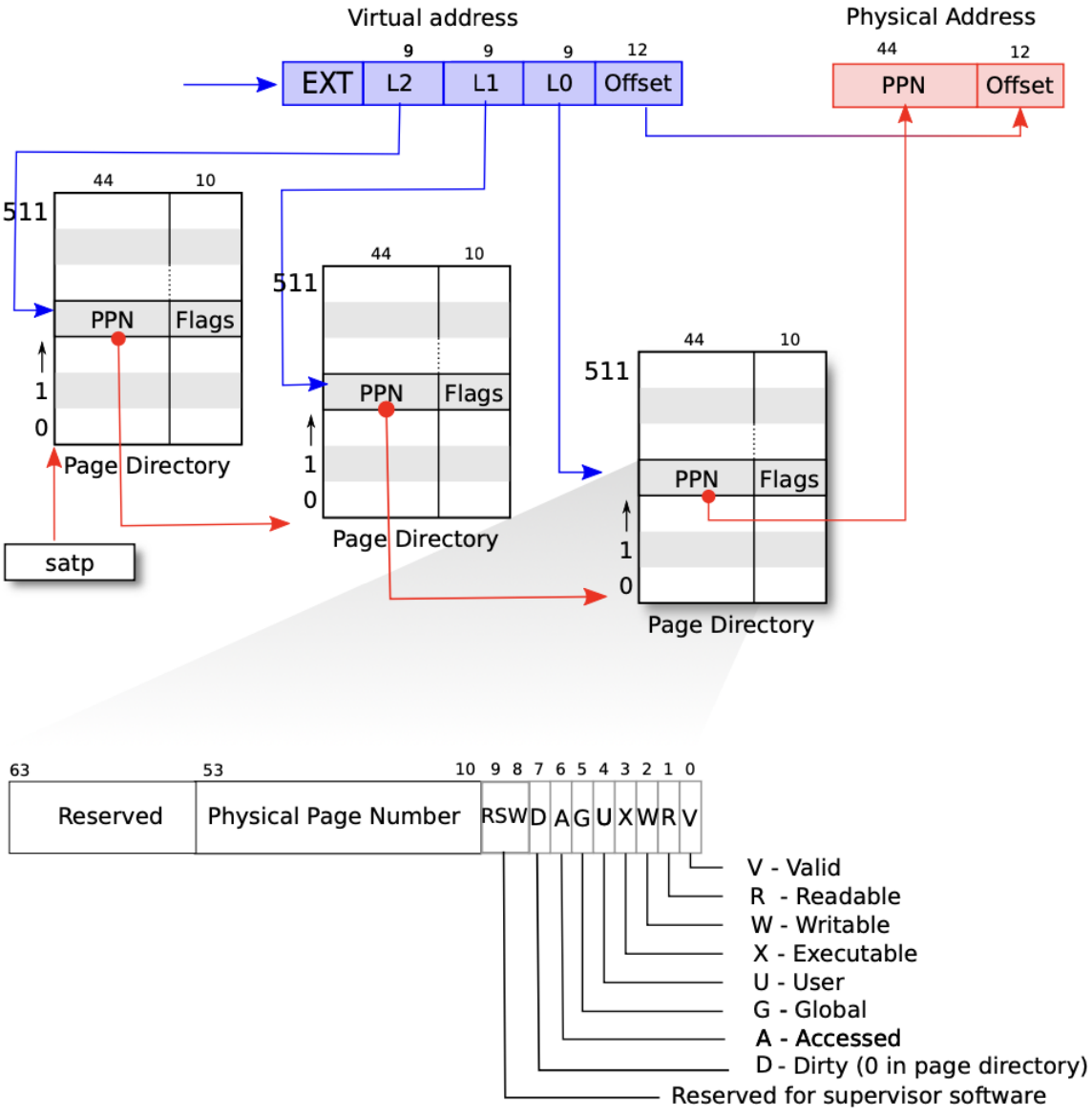

Figure 3.2: RISC-V address translation details.

In Sv39 RISC-V, the top 25 bits of a virtual address are not used for translation. 2^{39} bytes is 512 GB, which should be enough address space for applications

running on RISC-V computers. As Figure 3.2 shows, a RISC-V CPU translates a virtual address into a physical in three steps. A page table is stored in physical memory as a three-level tree. The root of the tree is a 4096-byte page table page that contains 512 PTEs, which contain the physical addresses for page-table pages in the next level of the tree. Each of those pages contains 512 PTEs for the final level in the tree. The paging hardware uses the top 9 bits of the 27 bits to select a PTE in the root page-table page, the middle 9 bits to select a PTE in a page-table page in the next level of the tree, and the bottom 9 bits to select the final PTE.

If any of the three PTEs required to translate an address is not present, the paging hardware raises a ***pagefault*** *exception*, leaving it up to the kernel to handle the exception.

Each PTE contains flag bits that tell the paging hardware how the associated virtual address is allowed to be used. `PTE_V` indicates whether the PTE is present: if it is not set, a reference to the page causes an exception (i.e., is not allowed). `PTE_R` controls whether instructions are allowed to read to the page. `PTE_W` controls whether instructions are allowed to write to the page. `PTE_X` controls whether the CPU may interpret the content of the page as instructions and execute them. `PTE_U` controls whether instructions in user mode are allowed to access the page; if `PTE_U` is not set, the PTE can be used only in supervisor mode.

A few notes about terms. Physical memory refers to storage cells in DRAM. A byte of physical memory has an address, called a physical address. Instructions use only virtual addresses, which the paging hardware translates to physical addresses, and then sends to the DRAM hardware to read 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.

A Brief Preview about the Project 3

You are expected to implement the `mmap` and `munmap` system calls in the xv6 kernel.

The `mmap` and `munmap` system calls allow UNIX programs to exert detailed control over their address spaces. They can be used to share memory among processes,

to map files into process address spaces, and as part of user-level page fault schemes such as the garbage-collection algorithms discussed in lecture. In this lab you'll add `mmap` and `munmap` to xv6, focusing on memory-mapped files.

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

`mmap` can be called in many ways, but this lab requires only a subset of its features relevant to memory-mapping a file. You can assume that `addr` will always be zero, meaning that the kernel should decide the virtual address at which to map the file. `mmap` returns that address, or 0xffffffffffffffff (i.e. -1) if it fails. length is the number of bytes to map; it might not be the same as the file's length. `prot` indicates whether the memory should be mapped readable, writeable, and/or executable; you can assume that `prot` is `PROT_READ` or `PROT_WRITE` or both. `flags` will be either `MAP_SHARED`, meaning that modifications to the mapped memory should be written back to the file, or `MAP_PRIVATE`, meaning that they should not. You don't have to implement any other bits in flags. `fd` is the open file descriptor of the file to map. It's OK if processes that map the same MAP_SHARED file do **not** share physical pages.

`munmap(addr, length)` should remove mmap mappings in the indicated address range. If the process has modified the memory and has it mapped `MAP_SHARED`, the modifications should first be written to the file. An `munmap` call might cover only a portion of an mmap-ed region, but you can assume that it will either `unmap` at the start, or at the end, or the whole region (but not punch a hole in the middle of a region).