Resumen libro xv6

Lautaro Bachmann

Contents

Operating system interfaces

The xv6 kernel provides a subset of the services and system calls that Unix kernels traditionally offer.

Processes and memory

An xv6 process

consists of user-space memory (instructions, data, and stack) and per-process state private to the kernel.

Xv6 time-shares processes:
it transparently switches the available CPUs among the set of processes waiting to execute.

When a process is not executing,
xv6 saves its CPU registers, restoring them when it next runs the process.

The kernel associates
a process identifier, or PID, with each process.

Xv6 allocates

most user-space memory implicitly: fork allocates the memory required for the child's copy of the parent's memory, and exec allocates enough memory to hold the executable file.

A process that needs more memory at run-time
can call sbrk(n) to grow its data memory by n bytes; sbrk returns the location of the new memory.

I/O and File descriptors

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.

convention,
a process reads from file descriptor 0 (standard input), writes output to file descriptor 1 (standard output), and writes error messages to file descriptor 2 (standard error).

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

A newly allocated file descriptor

is always the lowestnumbered unused descriptor of the current process.

The second argument to open

consists of a set of flags, expressed as bits, that control what open does. The possible values are defined in the file control (fcntl) header O_RDONLY, O_WRONLY, O_RDWR, O_CREATE, and O_TRUNC, which instruct open to open the file for reading, or for writing, or for both reading and writing, to create the file if it doesn't exist, and to truncate the file to zero length.

each underlying file offset

is shared between parent and child.

The dup system call

duplicates an existing file descriptor, returning a new one that refers to the same underlying I/O object. Both file descriptors share an offset, just as the file descriptors duplicated by fork do.

Two file descriptors share an offset

if they were derived from the same original file descriptor by a sequence of fork and dup calls. Otherwise file descriptors do not share offsets, even if they resulted from open calls for the same file.

Pipes

A pipe

is a small kernel buffer exposed to processes as a pair of file descriptors, one for reading and one for writing.

If no data is available,

a read on a pipe waits for either data to be written or for all file descriptors referring to the write end to be closed; read blocks until it is impossible for new data to arrive

Pipes have at least four advantages

pipes automatically clean themselves up; pipes can pass arbitrarily long streams of data, pipes allow for parallel execution of pipeline stages, if you are implementing inter-process communication, pipes' blocking reads and writes are more efficient than the non-blocking semantics of files.

File system

The xv6 file system

provides data files, which contain uninterpreted byte arrays, and directories, which contain named references to data files and other directories. The directories form a tree, starting at a special directory called the root.

Paths that don't begin with /

are evaluated relative to the calling process's current directory,

system calls to create new files and directories:

mkdir
creates a new directory,

open
with the O_CREATE flag creates a new data file, creates a new device file.

mknod

Mknod

creates a special file that refers to a device. Associated with a device file are the major and minor device numbers (the two arguments to mknod), which uniquely identify a kernel device. When a process later opens a device file, the kernel

diverts read and write system calls to the kernel device implementation instead of passing them to the file system.

A file's name is distinct from the file itself;

the same underlying file, called an **inode,** can have multiple names, called **links.**

Each link

consists of an entry in a directory; the entry contains a file name and a reference to an **inode.**

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.

The fstat system call

retrieves information from the inode that a file descriptor refers to. It fills in a struct stat, defined in stat.h

The link system call

creates another file system name referring to the same inode as an existing file.

Each inode

is identified by a unique inode number.

The unlink system call

removes a name from the file system. The file's inode and the disk space holding its content **are only freed** when the file's link count is zero and no file descriptors refer to it.

Real world

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.

Xv6 does not provide a notion of users

or of protecting one user from another; in Unix terms, all xv6 processes run as root.

Operating system organization

Kernel organization

what part of the operating system should run in supervisor mode.

the entire operating system resides in the kernel, so that the implementations of all system calls run in supervisor mode. This organization is called a **monolithic kernel.** In this organization the entire operating system runs with full hardware privilege.

To reduce the risk of mistakes in the kernel,

OS designers can minimize the amount of operating system code that runs in supervisor mode, and execute the bulk of the operating system in user mode. This kernel organization is called a **microkernel.**

In a microkernel,

the kernel interface consists of a few low-level functions for starting applications, sending messages, accessing device hardware, etc. This organization allows the kernel to be relatively simple, as most of the operating system resides in user-level servers.

In the real world,

both monolithic kernels and microkernels are popular. For example, Linux has a monolithic kernel, although some OS functions run as user-level servers Operating systems such as Minix, L4, and QNX are organized as a microkernel with servers, and have seen wide deployment in embedded settings.

There is much debate
among developers of operating systems which organization is better, and there is no conclusive evidence one way or the other.

File bio.c console.c entry.S exec.c file.c fs.c kalloc.c kernelvec.S log.c main.c pipe.c plic.c printf.c proc.c sleeplock.c spinlock.c start.c string.c swtch.S syscall.c sysfile.c sysproc.c trampoline.S trap.c uart.c virtio_disk.c vm.c Description Disk block cache for the file system. Connect to the user keyboard and screen. Very first boot instructions. exec() system call. File descriptor support. File system. Physical page allocator. Handle traps from kernel, and timer interrupts. File system logging and crash recovery. Control initialization of other modules during boot. Pipes. RISC-V interrupt controller. Formatted output to the console. Processes and scheduling. Locks that yield the CPU. Locks that don't yield the CPU. Early machine-mode boot code. C string and byte-array library. Thread switching. Dispatch system calls to handling function. File-related system calls. Process-related system calls. Assembly code to switch between user and kernel.

C code to handle and return from traps and interrupts. Serial-port console device driver. Disk device driver. Manage page tables and address spaces.

Xv6 kernel source files.

Xv6

is implemented as a monolithic kernel, like most Unix operating systems.

Code: xv6 organization

The xv6 kernel source is in the kernel/ sub-directory. The source is divided into files, following a rough notion of modularity; The inter-module interfaces are defined in

defs.h (kernel/defs.h).

Process overview

The unit of isolation in xv6

is a process. The process abstraction prevents one process from wrecking or spying on another process's memory, CPU, file descriptors, etc. It also prevents a process from wrecking the kernel itself, so that a process can't subvert the kernel's isolation mechanisms.

To help enforce isolation,

the process abstraction provides the illusion to a program that it has its own private machine. A process provides a program with what appears to be a private memory system, or address space, which other processes cannot read or write. A process also provides the program with what appears to be its own CPU to execute the program's instructions.

Xv6 uses page tables

(which are implemented by hardware) to give each process its own address space. The RISC-V page table translates a virtual address to a physical address Xv6 maintains **a separate page table** for each process that defines that process's address space.

an address space includes

the process's user memory starting at virtual address zero. Instructions come first, followed by global variables, then the stack, and finally a "heap" area that the process can expand as needed.

the maximum size of a process's address space:

pointers on the RISC-V are 64 bits wide; the hardware only uses the low 39 bits when looking up virtual addresses in page tables; and xv6 only uses 38 of those 39 bits. **Thus, the maximum address is 238 1**

At the top of the address space

xv6 reserves a page for a **trampoline** and a page mapping the process's **trapframe.**

Xv6 uses these two pages
to transition into the kernel and back;

the trampoline page
contains the code to transition in and out of the kernel and mapping the trapframe is necessary to save/restore the state of the user process,

pieces of state for each process,

which it gathers into a struct proc A process's most important pieces of kernel state are its page table, its kernel stack, and its run state.

Each process has

thread of execution
executes the process's instructions. A thread can be suspended and later resumed.

state of a thread
(local variables, function call return addresses) is stored on the thread's stacks.

Each process has two stacks:

user stack
When the process is executing user instructions, only its user stack is in use, and its kernel stack is empty.

When the process enters the kernel
the kernel code executes on the process's kernel stack; while a process is in the kernel, its user stack still contains saved data, but isn't actively used.

The kernel stack
is separate (and protected from user code) so that the kernel can execute even if a process has wrecked its user stack.

p->state

indicates whether the process is allocated, ready to run, running, waiting for I/O, or exiting.

p->pagetable

holds the process's page table, in the format that the RISC-V hardware expects.

Xv6 causes the paging hardware
to use a process's **p->pagetable** when executing that process in user space.

Code: starting xv6, the first process and system call

When the RISC-V computer powers on,

it initializes itself
and runs a boot loader which is stored in read-only memory. The boot loader loads the xv6 kernel into memory.

Then, in machine mode,
the CPU executes xv6 starting at _entry

The RISC-V starts
with paging hardware disabled: virtual addresses map directly to physical addresses.

The loader loads the xv6 kernel
into memory at physical address 0x80000000. The reason it places the kernel at 0x80000000 rather than 0x0 is because the address range 0x0:0x80000000 contains I/O devices.

The instructions at _entry
set up a stack so that xv6 can run C code. Xv6 declares space for an initial stack, stack0, in the file start.c The code at _entry loads the stack pointer register sp with the address stack0+4096, **the top of the stack,** because the stack on RISC-V **grows down.** Now that the kernel has a stack, _entry calls into C code at start

The function start
performs some configuration that is only allowed in machine mode, and then switches to supervisor mode. Before jumping into supervisor mode, it programs the clock chip to generate **timer interrupts.** With this out of the way, **start** "returns" to supervisor mode

change to main

After **main** initializes several devices and subsystems, it creates the first process by calling **userinit** The first process executes a small program written in RISC-V assembly, which makes the first system call in xv6. **initcode.S (user/initcode.S:3)** loads the number for the exec system and then re-enter the kernel.

Once the kernel has completed exec,

it returns to user space in the /init process. **Init**

creates a new console device file if needed and then opens it as file descriptors 0, 1, and 2. Then it starts a shell on the console. **The system is up.**

Real world

Most operating systems have adopted the process concept, and most processes look similar to xv6's. Modern operating systems, however, support several threads within a process, to allow a single process to exploit multiple CPUs. Supporting multiple threads in a process involves quite a bit of machinery that **xv6 doesn't have,**