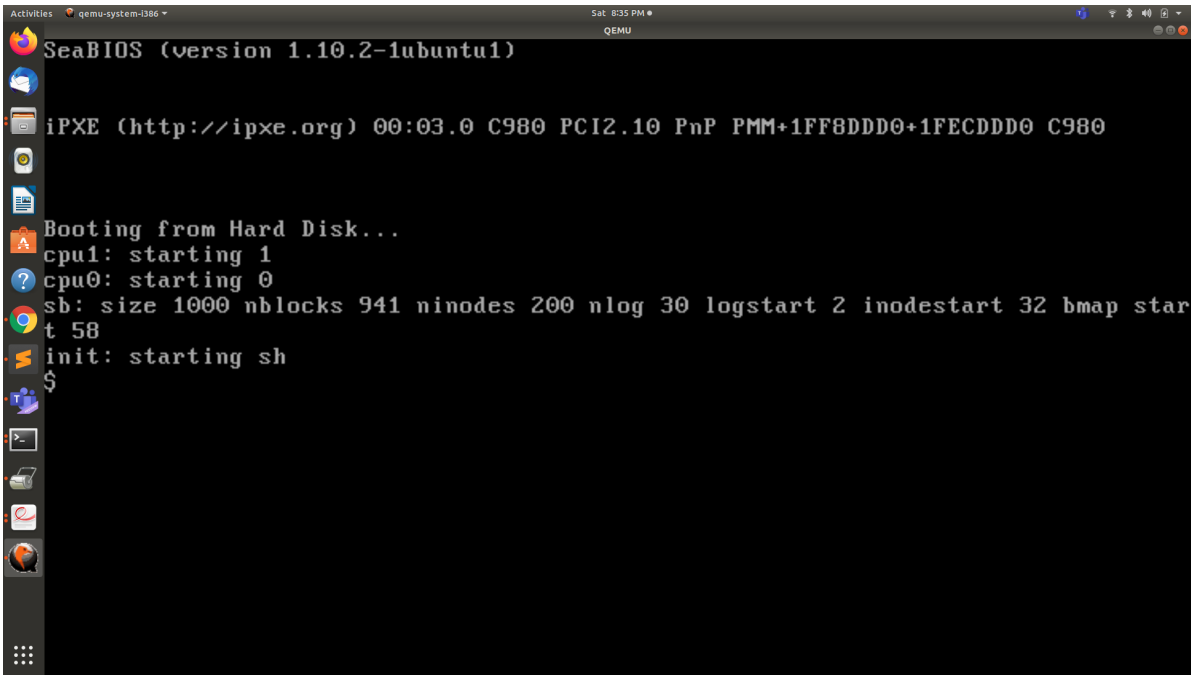
xv6 : Operating System

Introduction

Xv6 is a teaching operating system developed in the summer of 2006 for MIT's operating systems course . A sample of xv6 System.



History

xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix Version 6 (v6). xv6 loosely follows the structure and style of v6, but is implemented for a modern x86-based multiprocessor using ANSI C.

xv6 borrows code from the following sources:

```
JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, and others)
    Plan 9 (entryother.S, mp.h, mp.c, lapic.c)
    FreeBSD (ioapic.c)
    NetBSD (console.c)
```

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run
"make". On non-x86 or non-ELF machines (like OS X, even on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries

A copy of that version of xv6 is available via

```
git clone https://github.com/uva-reiss-cs4414/xv6.git
```

Operating System Interfaces

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. When a process needs to invoke a kernel service, it invokes a procedure call in the operating system interface. Such a procedure is called a system call. 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 kernel uses the CPU's hardware protection mechanisms to ensure that each process executing in user space can access only its own memory. The kernel executes with the hardware privileges required to implement these protections; user programs execute without those privileges. When a user program invokes a system call, the hardware raises the privilege level and starts executing a pre-arranged function in the kernel. 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.

Processes and memory..

An xv6 process consists of user-space memory (instructions, data, and stack) and per-process state private to the kernel. Xv6 can time-share 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.You're insecure Don't know what for You're turning heads when you walk through the door Don't need make-up to cover up Being the way that you are is enough Everyone else in the room can see it Everyone else but you Baby, you light up my world like nobody else The way that you flip your hair gets me overwhelmed But when you smile at the ground it ain't hard to tell You don't know, oh oh You don't know you're beautiful If only you saw what I can see You'll understand why I want you so desperately Right now I'm looking at you and I can't believe You don't know, oh oh You don't know you're beautiful, oh oh That's what makes you beautiful So co-come on, you got it wrong To prove I'm right, I put it in a song I don't know why, you're being shy The kernel associates a process identifier, or pid, with each process. A process may create a new process using the fork system call. Fork creates a new process, called the child process, with exactly the same memory contents as the calling process, called the parent process. Fork returns in both the parent and the child. In the parent, fork returns the child's pid; in the child, it returns zero. For example, consider the following program fragment:

```
int pid = fork();
if(pid > 0){
printf("parent: child=%d\n", pid);
pid = wait();
printf("child %d is done\n", pid);
} else if(pid == 0){
printf("child: exiting\n");
exit();
} else {
printf("fork error\n");
}
```

The exit system call causes the calling process to stop executing and to release re

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 (perhaps for malloc) can call sbrk(n) to grow its data memory by n bytes; sbrk returns the location of the new memory. Xv6 does not provide a notion of users or of protecting one user from another; in Unix terms, all xv6 processes run as root.

I/O and File descriptors..

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. By 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). As we will see, the shell exploits the convention to implement I/O redirection and pipelines. You're insecure Don't know what for You're turning heads when you walk through the door Don't need make-up to cover up Being the way that you are is enough Everyone else in the room can see it Everyone else but you Baby, you light up my world like nobody else The way that you flip your hair gets me overwhelmed But when you smile at the ground it ain't hard to tell You don't know, oh oh You don't know you're beautiful If only you saw what I can see You'll understand why I want you so desperately Right now I'm looking at you and I can't believe You don't know, oh oh You don't know you're beautiful, oh oh That's what makes you beautiful So co-come on, you got it wrong To prove I'm right, I put it in a song I don't know why, you're being shyThe shell ensures that it always has three file descriptors open (8707), which are by default file descriptors for the console. 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 signal 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.

```
char buf[512];
int n;
for(;;){
n = read(0, buf, sizeof buf);
if(n == 0)
break;
if(n < 0){
fprintf(2, "read error\n");
exit();
}
if(write(1, buf, n) != n){
fprintf(2, "write error\n");
exit();
}
```

```
}
```

Processes and memory

An xv6 process consists of user-space memory (instructions, data, and stack) and per-process state private to the kernel. Xv6 can time-share 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.
A process may create a new process using the fork system call. Fork creates a new process, called the child process, with exactly the same memory contents as the calling process, called the parent process. Fork returns in both the parent and the child. In the parent, fork returns the child's pid; in the child, it returns zero. For example, consider the following program fragment:

```
int pid = fork();
if(pid > 0){
printf("parent: child=%d\n", pid);
pid = wait();
printf("child %d is done\n", pid);
} else if(pid == 0){
printf("child: exiting\n");
exit();
} else {
printf("fork error\n");
}
```

fork()

Every process has a process ID (or pid for short).

A process can call the kernel to do `fork()`, which creates a new process, which is entirely identical to the parent process (registers, memory and everything). The only differences are:

- The pid
- The value returned from fork:
- In the new process - 0
- In the parent process - the new pid
- In case of failure - some negative error code

exec()

`Fork()` creates a new process, and leaves the parent running. `Exec()`, on the other hand, replaces the process's program with a new program. It's still the same process, but with new code (and variables, stack, etc.). Registers, pid, etc. remain the same.

It is common practice to have the child of `fork` call `exec` after making sure it is the child. So why not just make a single function that does both fork and exec together? The exec system call replaces the entire memory of the parent process except for the open files.Take me down to the river bend Take me down to the fighting end Wash the poison from off my skin Show me how to be whole again Fly me up on a silver wing Past the black where the sirens sing Warm me up in a nova's glow And drop me down to the dream below 'Cause I'm only a crack in this castle of glass Hardly anything there for you to see For you to see Bring me home in a blinding dream Through

the secrets that I have seen Wash the sorrow from off my skin Show me how to be whole again This allows a parent process to decide what the stdin, stdout and stderr for the child process. This trick is used in the /init process in xv6 and the sh to pipe outputs to other processes.

Process termination

Trigger warning: sad stuff ahead. And also zombies.

A process will be terminated if (and only if) one of the following happens:

1. The process invokes `exit()`
2. Some other process invokes `kill()` with its pid
3. The process generates some exception

Note that `kill` does not actually terminate the process. What it does is leave a mark of "You need to kill yourself" on the process, and it'll be the process itself that commits suicide (after it starts running again, when the scheduler loads it).

Note also that not any process can `kill` any other process. The kernel makes sure of that.

Once `kill`ed, the process's resources (memory, etc.) are not released yet,
until its parent (that is, the process which called for its creation) allows this to happen.
A process that `kill`ed itself but whose parent did not acknowledge this is called a zombie.

In order to a parent to "acknowledge" its child's termination, it needs to call `wait()`.
When it calls `wait()`, it will not continue until one of its children exits, and then it will continue.
If there are a few children, the parent will need to call `wait` once for each child process.

`Wait` returns the pid of the exited process.

What happens if a parent process `exit`s before its children?
Its children become orphans, and the The First Process (whose pid is 1) will make them His children.

Kernel organization..

A key design question for an operating system is what part of the operating system
should run in kernel mode. A simple answer is that the kernel interface is the system
call interface. That is, fork, exec, open, close, read, write, etc. are all kernel
calls. This choice means that the complete implementation of the operating system
runs in kernel mode. This kernel organization is called a monolithic kernel.
In this organization the complete operating system runs with full hardware privilege.
This organization is convenient because the OS designer doesn’t have to decide
which part of the operating system doesn’t need full hardware privilege. Furthermore,
it easy for different parts of the operating system to cooperate. For example, an operating
system might have a buffer cache that can be shared both by the file system and
the virtual memory system. Take me down to the river bend Take me down to the fighting end Wash the poison from off my skin Show me how to be whole again Fly me up on a silver wing Past the black where the sirens sing Warm me up in a nova's glow And drop me down to the dream below 'Cause I'm only a crack in this castle of glass Hardly anything there for you to see For you to see Bring me home in a blinding dream Through the secrets that I have seen Wash the sorrow from off my skin Show me how to be whole again
A downside of the monolithic organization is that the interfaces between different
parts of the operating system are often complex (as we will see in the rest of this text),
and therefore it is easy for an operating system developer to make a mistake. In a
monolithic kernel, a mistake is fatal, because an error in kernel mode will often result

in the kernel to fail. If the kernel fails, the computer stops working, and thus all applications fail too. The computer must reboot to start again.

Paging hardware

As a reminder, x86 instructions (both user and kernel) manipulate virtual addresses. The machine's RAM, or physical memory, is indexed with physical addresses. The x86 page table hardware connects these two kinds of addresses, by mapping each virtual address to a physical address.
An x86 page table is logically an array of 2^20 (1,048,576) page table entries (PTEs). Each PTE contains a 20-bit physical page number (PPN) and some flags. The paging hardware translates a virtual address by using its top 20 bits to index into the page table to find a PTE, and replacing the address's top 20 bits with the PPN in the PTE. The paging hardware copies the low 12 bits unchanged from the virtual to the translated physical address. Thus 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.

Process address space

The page table created by entry has enough mappings to allow the kernel's C code to start running. However, main immediately changes to a new page table by calling kvmalloc (1857), because kernel has a more elaborate plan for describing process address spaces.
Each process has a separate page table, and xv6 tells the page table hardware to switch page tables when xv6 switches between processes. As shown in Figure 2-2, a process's user memory starts at virtual address zero and can grow up to KERNBASE, allowing a process to address up to 2 GB of memory.

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 (6310) opens the named binary path using namei (6321), which is explained in Chapter 6. Then, it reads the ELF header. Xv6 applications are described in the widely-used ELF format, defined in elf.h. An ELF binary consists of an ELF header, struct elfhdr (0955), followed by a sequence of program section headers, struct proghdr (0974). 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.

Code: C trap handler

We saw in the last section that each handler sets up a trap frame and then calls the C function trap. Trap (3351) looks at the hardware trap number tf->trapno to decide why it has been called and what needs to be done. If the trap is T_SYSCALL, trap calls the system call handler syscall. We'll revisit the two proc->killed checks in .

After checking for a system call, trap looks for hardware interrupts (which we discuss below). In addition to the expected hardware devices, a trap can be caused by a spurious interrupt, an unwanted hardware interrupt.
If the trap is not a system call and not a hardware device looking for attention, trap assumes it was caused by incorrect behavior (e.g., divide by zero) as part of the

code that was executing before the trap. If the code that caused the trap was a user program, xv6 prints details and then sets cp->killed to remember to clean up the user process. We will look at how xv6 does this cleanup in Chapter 5.
If it was the kernel running, there must be a kernel bug: trap prints details about the surprise and then calls panic.

Code: System calls

For system calls, trap invokes syscall (3625). Syscall loads the system call number from the trap frame, which contains the saved %eax, and indexes into the system call tables. For the first system call, %eax contains the value SYS_exec (3457), and syscall will invoke the SYS_exec'th entry of the system call table, which corresponds to invoking sys_exec.
Syscall records the return value of the system call function in %eax. When the trap returns to user space, it will load the values from cp->tf into the machine registers. Thus, when exec returns, it will return the value that the system call handler returned (3631). System calls conventionally return negative numbers to indicate errors, positive numbers for success. If the system call number is invalid, syscall prints an error and returns –1.