Porting xv6 to Loongson 3A, a CPU in MIPS64 family

Gan Quan

April 9, 2015

Contents

Chapter 1

Operating system interfaces

The job of an operating system is to share a computer among multiple programs and to provide a more useful set of services than the hardware alone supports. The operating system manages and abstracts the low-level hardware, so that, for example, a word processor need not concern itself with which type of disk hardware is being used. It also multiplexes the hardware, allowing many programs to share the computer and run (or appear to run) at the same time. Finally, operating systems provide controlled ways for programs to interact, so that they can share data or work together.

An operating system provides services to user programs through an interface. Designing a good interface turns out to be difficult. On the one hand, we would like the interface to be simple and narrow because that makes it easier to get the implementation right. On the other hand, we may be tempted to offer many sophisticated features to applications. The trick in resolving this tension is to design interfaces that rely on a few mechanisms that can be combined to provide much generality.

This book uses a single operating system as a concrete example to illustrate operating system concepts. That 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, Mac OS X, 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.

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 (with arguments)	Description
`fork()`	Create process
`exit(code)`	Terminate current process
`waitpid(pid, stat, opts)`	Wait for a child process to exit
`kill(pid, sig)`	Send signal to a process with given PID
`getpid()`	Get current process ID
`sleep(n)`	Sleep for `n` seconds
`usleep(n)`	Sleep for (about) `n` microseconds
`execve(name, argv, envp)`	Load and execute a file
`sbrk(incr)`	Grow or shrink process heap
`open(name, flags)`	Open a file or device in some mode
`read(fd, buf, len)`	Read from opened file or device
`write(fd, buf, len)`	Write to opened file or device
`close(fd)`	Close an opened file
`dup(fd)`	Duplicate a file descriptor
`dup2(old, new)`	Duplicate a file descriptor with preference
`pipe(p)`	Create a pipe
`chdir(dir)`	Change current directory
`mkdir(dir)`	Make directory
`mknod(name, major, minor)`	Make a device file
`fstat(fd, stat)`	Fetch file info
`link(old, new)`	Create another name for given file
`unlink(name)`	Remove a name or a file

Figure 1.1: List of system calls supported by xv6', with difference from xv6 colored red

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

The current implementation, dubbed xv6', (is planned to) add some more features and functions provided my more modern operating systems. The system calls supported by xv6' is listed in Figure1.1. Currently, only `write()` is implemented, and the `write()` call can only print contents to a serial.

The rest of this chapter outlines xv6's services–processes, memory, file descriptors, pipes, and file system–and illustrates them with code snippets and discussions of how the shell uses them. The shell's use of system calls illustrates

how carefully they have been designed.

The shell is an ordinary program that reads commands from the user and executes them, and is the primary user interface to traditional Unix-like systems. The fact that the shell is a user program, not part of the kernel, illustrates the power of the system call interface: there is nothing special about the shell. It also means that the shell is easy to replace; as a result, modern Unix systems have a variety of shells to choose from, each with its own user interface and scripting features. The xv6 shell is a simple implementation of the essence of the Unix Bourne shell. The shell is not implemented yet.

1.1 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();
int status;
if(pid > 0){
        printf("parent: child=%d\n", pid);
        waitpid(pid, &status, 0);
        printf("child %d is done\n", pid);
} else if(pid == 0){
        printf("child: exiting\n");
        exit(0);
} else {
        printf("fork error\n");
}
```

The exit system call causes the calling process to stop executing and to release resources such as memory and open files. The `waitpid` system call returns the pid of an exited child of the current process; if none of the caller's children has exited, `waitpid` waits for one to do so. In the example, the output lines

```
parent: child=1234
child: exiting
```

might come out in either order, depending on whether the parent or child gets to its `printf` call first. After the child exits the parent's wait returns, causing the parent to print

```
parent: child 1234 is done
```

Note that the parent and child were executing with different memory and different registers: changing a variable in one does not affect the other.

The `execve` system call replaces the calling process's memory with a new memory image loaded from a file stored in the file system. The file must have a particular format, which specifies which part of the file holds instructions, which part is data, at which instruction to start, etc. xv6 uses the ELF format, which Chapter 3 discusses in more detail. When `execve` succeeds, it does not return to the calling program; instead, the instructions loaded from the file start executing at the entry point declared in the ELF header. `execve` takes two arguments: the name of the file containing the executable and an array of string arguments. For example:

```
char *argv[3];
argv[0] = "echo";
argv[1] = "hello";
argv[2] = 0;
execve("/bin/echo", argv, NULL);
printf("exec error\n");
```

This fragment replaces the calling program with an instance of the program `/bin/echo` running with the argument list `echo hello`. Most programs ignore the first argument, which is conventionally the name of the program.

The xv6 shell uses the above calls to run programs on behalf of users. The main structure of the shell is simple. The main loop reads the input on the command line using `getcmd`. Then it calls `fork`, which creates a copy of the shell process. The parent shell calls `waitpid`, while the child process runs the command. For example, if the user had typed "echo hello" at the prompt, `runcmd` would have been called with `"echo hello"` as the argument. `runcmd` runs the actual command. For `"echo hello"`, it would call `execve`. If `execve` succeeds then the child will execute instructions from `echo` instead of `runcmd`. At some point `echo` will call `exit`, which will cause the parent to return from `waitpid` in main. You might wonder why `fork` and `execve` are not combined in a single call; we will see later that separate calls for creating a process and loading a program is a clever design.

Xv6 allocates most user-space memory implicitly: `fork` allocates the memory required for the child's copy of the parent's memory, and `execve` 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. If there's still time, perhaps *user access control* could be added into xv6'?

1.2 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, `STDIN_FILENO`), writes output to file descriptor 1 (standard output, `STDOUT_FILENO`), and writes error messages to file descriptor 2 (standard error, `STDERR_FILENO`). As we will see, the shell exploits the convention to implement I/O redirection and pipelines. The shell ensures that it always has three file descriptors open, 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.

The following program fragment (which forms the essence of `cat`) copies data from its standard input to its standard output. If an error occurs, it writes a message to the standard error.

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

The important thing to note in the code fragment is that `cat` doesn't know whether it is reading from a file, console, or a pipe. Similarly `cat` doesn't know whether it is printing to a console, a file, or whatever. The use of file descriptors and the convention that file descriptor 0 is input and file descriptor 1 is output allows a simple implementation of `cat`.

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.

File descriptors and `fork` interact to make I/O redirection easy to implement. `fork` copies the parent's file descriptor table along with its memory, so that the child starts with exactly the same open files as the parent. The system call `execve` replaces the calling process's memory but preserves its file table. This behavior allows the shell to implement I/O redirection by forking, reopening chosen file descriptors, and then `execve`-ing the new program. Here is a simplified version of the code a shell runs for the command `cat <input.txt`:

```
char *argv[2];
argv[0] = "cat";
argv[1] = NULL;
if(fork() == 0) {
        fd = open("input.txt", O_RDONLY);
        dup2(fd, STDIN_FILENO);
        close(fd);
        execve("cat", argv, NULL);
}
```

Unlike original xv6, child process in this code snippet opens `input.txt` in a new file descriptor, which is copied to the standard input file descriptor (see below), closing the standard input during the process (as specified by `dup2(2)` in POSIX standard). The child then invokes `execve`, with the newly-allotted file descriptor closed, but the (duplicated) standard input preserved. This snippet somehow resembles more to modern programs.

The code for I/O redirection in the xv6 shell works in exactly this way. Recall that at this point in the code the shell has already forked the child shell and that `runcmd` will call `execve` to load the new program. Now it should be clear why it is a good idea that `fork` and `execve` are separate calls. This separation allows the shell to fix up the child process before the child runs the intended program.

Although fork copies the file descriptor table, each underlying file offset is shared between parent and child. Consider this example:

```
if((pid = fork()) == 0) {
        write(STDOUT_FILENO, "hello ", 6);
        exit(0);
} else {
        waitpid(pid, &status, 0);
        write(STDOUT_FILENO, "world\n", 6);
}
```

At the end of this fragment, the file attached to file descriptor 1 will contain the data hello world. The `write` in the parent (which, thanks to `waitpid`, runs only after the child is done) picks up where the child's `write` left off. This behavior helps produce sequential output from sequences of shell commands, like `(echo hello; echo world) >output.txt`.

The `dup` (and `dup2` in xv6') 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. This is another way to write `hello world` into a file:

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

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. `dup` allows shells to implement commands like this: `ls existing-file non-existing-file > tmp1 2>&1`. The `2>&1` tells the shell to give the command a file descriptor 2 that is a duplicate of descriptor 1. Both the name of the existing file and the error message for the non-existing file will show up in the file `tmp1`. The xv6 shell doesn't support I/O redirection for the error file descriptor, but now you know how to implement it. Standard error redirection would be implemented in xv6'.

File descriptors are a powerful abstraction, because they hide the details of what they are connected to: a process writing to file descriptor 1 may be writing to a file, to a device like the console, or to a pipe.

1.3 Pipes

A *pipe* is a small kernel buffer exposed to processes as a pair of file descriptors, one for reading and one for writing. Writing data to one end of the pipe makes that data available for reading from the other end of the pipe. Pipes provide a way for processes to communicate, that is, it is a form of *inter-process communication (IPC)*. Other forms of IPC include *shared memory* and *signals*, which would be implemented (potentially much) later in xv6'.

The following example code runs the program `wc` with standard input connected to the read end of a pipe.

```
int p[2];
char *argv[2];
argv[0] = "wc";
argv[1] = NULL;
pipe(p);
if(fork() == 0) {
        dup2(p[0], STDIN_FILENO);
        close(p[0]);
        close(p[1]);
        execve("/bin/wc", argv, NULL);
} else {
        write(p[1], "hello world\n", 12);
        close(p[0]);
        close(p[1]);
}
```

The program calls `pipe`, which creates a new pipe and records the read and write file descriptors in the array `p`. After `fork`, both parent and child have file descriptors referring to the pipe. The child `dup`s the read end onto file

descriptor 0, closes the file descriptors in `p`, and `execve`s `wc`. When `wc` reads from its standard input, it reads from the pipe. The parent writes to the write end of the pipe and then closes both of its file descriptors.

If no data is available, a `read` on a pipe waits for either data to be written or all file descriptors referring to the write end to be closed; in the latter case, read will return 0, just as if the end of a data file had been reached. The fact that `read` blocks until it is impossible for new data to arrive is one reason that it's important for the child to close the write end of the pipe before executing `wc` above: if one of `wc`'s file descriptors referred to the write end of the pipe, wc would never see end-of-file.

The xv6 shell implements pipelines such as `grep fork sh.c | wc -l` in a manner similar to the above code. The child process creates a pipe to connect the left end of the pipeline with the right end. Then it calls `runcmd` for the left end of the pipeline and `runcmd` for the right end, and waits for the left and the right ends to finish, by calling `waitpid` twice. The right end of the pipeline may be a command that itself includes a pipe (e.g., `a | b | c`), which itself forks two new child processes (one for `b` and one for `c`). Thus, the shell may create a tree of processes. The leaves of this tree are commands and the interior nodes are processes that wait until the left and right children complete. In principle, you could have the interior nodes run the left end of a pipeline, but doing so correctly would complicate the implementation.

Pipes may seem no more powerful than temporary files: the pipeline

```
echo hello world | wc
```

could be implemented without pipes as

```
echo hello world >/tmp/xyz; wc </tmp/xyz
```

There are at least three key differences between pipes and temporary files. First, pipes automatically clean themselves up; with the file redirection, a shell would have to be careful to remove `/tmp/xyz` when done. Second, pipes can pass arbitrarily long streams of data, while file redirection requires enough free space on disk to store all the data. Third, pipes allow for synchronization: two processes can use a pair of pipes to send messages back and forth to each other, with each read blocking its calling pro- cess until the other process has sent data with `write`.

1.4 File system

As long as my box doesn't have a hard disk or other external storage, I have to implement a file system like `tmpfs`. Nevertheless, a file system should be implemented so as to demonstrate a key operating system concept. Kernel file systems such as `procfs` or `sysfs` could also be implemented, if I have time. Another workaround is to create some kind of *RAM disk*. It is preferred to implement a maybe-simplified version of contemporary file system, like Berkeley FFS, if time permitting.

The xv6 file system provides data files, which are uninterpreted byte arrays, and directories, which contain named references to data files and other directories. Xv6 implements directories as a special kind of file. The directories form a tree, starting at a special directory called the *root*. A path like `/a/b/c` refers to the file or directory named `c` inside the directory named `b` inside the directory named `a` in the root directory `/`. Paths that don't begin with `/` are evaluated relative to the calling process's current directory, which can be changed with the `chdir` system call. Both these code fragments open the same file (assuming all the directories involved exist):

```
chdir("/a");
chdir("b");
open("c", O_RDONLY);

open("/a/b/c", O_RDONLY);
```

The first fragment changes the process's current directory to `/a/b`; the second neither refers to nor modifies the process's current directory.

There are multiple system calls to create a new file or directory: `mkdir` creates a new directory, `open` with the `O_CREATE` flag creates a new data file, and `mknod` creates a new device file. This example illustrates all three:

```
mkdir("/dir");
fd = open("/dir/file", O_CREATE|O_WRONLY);
close(fd);
mknod("/console", 1, 1);
```

`mknod` creates a file in the file system, but the file has no contents. Instead, the file's metadata marks it as a device file and records the major and minor device numbers (the two arguments to `mknod`), which uniquely identify a kernel device. When a process later opens the file, the kernel diverts `read` and `write` system calls to the kernel device implementation instead of passing them to the file system.

`fstat` retrieves information about the object a file descriptor refers to. It fills in a `struct stat`, defined in `stat.h` as `stat` structure may become different from original xv6 implementation. :

```
#define T_DIR 1
#define T_FILE 2
#define T_DEV 3
struct stat {
        short type;
        int dev;
        uint ino;
        short nlink;
        uint size;
};
```

A file's name is distinct from the file itself; the same underlying file, called an `inode`, can have multiple names, called *links*. The `link` system call creates another file system name referring to the same `inode` as an existing file. This fragment creates a new file named both `a` and `b`.

```
open("a", O_CREATE|O_WRONLY);
link("a", "b");
```

Reading from or writing to `a` is the same as reading from or writing to `b`. Each inode is identified by a unique *inode number*. After the code sequence above, it is possible to determine that a and b refer to the same underlying contents by inspecting the result of `fstat`: both will return the same inode number (`ino`), and the `nlink` count will be set to 2.

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. Thus adding

```
unlink("a");
```

to the last code sequence leaves the inode and file content accessible as b. Further- more,

```
fd = open("/tmp/xyz", O_CREATE|O_RDWR);
unlink("/tmp/xyz");
```

is an idiomatic way to create a temporary inode that will be cleaned up when the process closes `fd` or exits.

Xv6 commands for file system operations are implemented as user-level programs such as `mkdir`, `ln`, `rm`, etc. This design allows anyone to extend the shell with new user commands. In hind-sight this plan seems obvious, but other systems designed at the time of Unix often built such commands into the shell (and built the shell into the kernel).

One exception is `cd`, which is built into the shell. `cd` must change the current working directory of the shell itself. If `cd` were run as a regular command, then the shell would fork a child process, the child process would run `cd`, and `cd` would change the child's working directory. The parent's (i.e., the shell's) working directory would not change.

Chapter 2

Operating system organization

A key requirement for an operating system is to support several activities. For example, using the system call interface described in chapter 0 a process can start new processes using `fork`. The operating system must arrange that these processes can *time-share* the resources of the computer. For example, a process may start more new processes than there are processors in the computer, yet all processes must be able to make some progress. In addition, the operating system must arrange for *isolation* between the processes. That is, if one process has a bug and fails, it shouldn't impact processes that don't have a dependency on the failed process. Complete isolation, however, is too strong, since it should be possible for processes to interact; for example, it is convenient for users to combine processes to perform complex tasks (e.g., by using pipes). Thus, the implementation of an operating system must achieve three require- ments: multiplexing, isolation, and interaction.

This chapter provides an overview of how operating systems are organized to achieve these 3 requirements. It turns out there are many ways to do so, but this text focuses on mainstream designs centered around a *monolithic kernel*, which is used by many Unix operating systems. This chapter illustrates this organization by tracing the first process that is created when xv6 starts running. In doing so, the text provides a glimpse of the implementation of all major abstractions that xv6 provides, how they interact, and how the three requirements of multiplexing, isolation, and interaction are met. Most of xv6 avoids special-casing the first process, and instead reuses code that xv6 must provide for standard operation. Subsequent chapters will explore each ab- straction in more detail.

Xv6 runs on Intel 80386 or later ("x86") processors on a PC platform, and much of its low-level functionality (for example, its process implementation) is x86-specific. This book assumes the reader has done a bit of machine-level programming on some architecture, and will introduce x86-specific ideas as they come up. Appendix A briefly outlines the PC platform.

The original xv6 implementation mixes hardware-specific code and hardware-independent ones together. xv6' primarily aims to port xv6 from x86 architecture onto Loongson 3A, a MIPS64 CPU, in a so-called Loongson Multitech Board. Xv6' also tries to decouple these two parts, and potentially machine-dependent and architecture-dependent but machine-common codes.

2.1 Abstracting physical resources

The first question one might ask when encountering an operating system is why have it at all? That is, one could implement the system calls in Figure1.1 as a library, with which applications link. In this plan, each application could even have its own library, perhaps tailored to its needs. In this plan, the application can directly interact with the hardware resources and use those resources in the best way for the application (e.g., to achieve high performance or predictable performance). Some tiny operating systems for embedded devices or real-time systems are organized in this way.

The downside of this approach is that applications are free to use the library, which means they can also *not* use it. If they don't use the operating system library, then the operating system cannot enforce time sharing. It must rely on the application to behave properly and, for example, periodically give up a processor so that another application can run. Such a *cooperative* time-sharing scheme is maybe OK for a system where all applications trust each other, but doesn't provide strong isolation if applications are mutually distrustful.

To achieve strong isolation a helpful approach is to disallow applications to have direct access to the hardware resources, but instead to abstract the resources into services. For example, applications interact with a file system only through `open`, `read`, `write`, and `close` system calls, instead of read and writing raw disk sectors. This provides the application with the convenience of path names, and it allows the operating system (as the implementer of the interface) to manage the disk.

Similarly, in Unix applications run as processes using `fork`, allowing the operating system to save and restore registers on behalf of the application when switching between different processes, so that application don't have to be aware of process switching. Furthermore, it allows the operating system to forcefully switch an application out of a processor, if the application, for example, is an end-less loop.

As another example, Unix processes use `execve` to build up their memory image, instead of directly interacting with physical memory. This allows the operating system to decide where to place a process in memory and move things around if there is a shortage of memory, and provides applications with the convenience of a file system to store their images.

To support controlled interaction between applications, Unix applications can use only file descriptors, instead of to make up some sharing convention of their own (e.g., reserving a piece of physical memory). Unix file descriptors abstract all the sharing details away, hiding from the application if the interaction

is happening with the terminal, file system, or pipes, yet allows the operating system to control the interaction. For example, if one application fails, it can shut down the communication channel.

As you can see, the system call interface in Figure1.1 is carefully designed to provide programmer convenience but also for the implementation of the interface to enforce strong isolation. The Unix interface is not the only way to abstract resources, but it has proven to be a very good one.

2.2 User mode, kernel mode, and system calls

To provide strong isolation between the software that uses system calls and the software that implements the system calls, we need a hard boundary between applications and the operating system. If the application makes a mistake, we don't want the operating system to fail. Instead, the operating system should be able to clean up the application and continue running other applications. This strong isolation means that application shouldn't be able to write over data structures maintained by the operating system, shouldn't be able to overwrite instructions of the operating system, etc.

To provide for such strong isolation processors provide hardware support. For example, the x86 processor, like many other processors, has two modes in which the processor executes instructions: *kernel mode* and *user mode*. In kernel mode the processor is allowed to execute *privileged instructions*. For example, read and writing to the disk (or any other I/O device) is a privileged instruction. If an application in user mode attempts to execute a privileged instruction, then the processor doesn't execute the instruction, but switches to kernel mode so that the software in kernel mode can clean up the application, because it did something it shouldn't be doing. MIPS processors have a *supervisor mode* in addition to traditional user and kernel mode, but it is generally not used in modern operating system design. Applications can execute only user mode instructions (e.g., adding numbers, etc.) and is said to be running in *user space*, while the software in kernel mode can execute also privileged instructions and is said to be running in *kernel space*. The software running in kernel space (or in kernel mode) is called the *kernel*.

If a user-mode application must read or write to disk, it must transition to the kernel to do so, because the application itself can not execute I/O instructions. Processors provide a special instruction that switches the processor from user mode to kernel mode and enters the kernel at an entry point specified by the kernel. (The x86 processor provides the int instruction for this purpose, whereas the MIPS processors propose various instructions like `syscall`, `break`, or trap instructions, the `syscall` instruction mostly used.) Once the processor has switched to kernel mode, the kernel can then validate the arguments of the system call, decide whether the application is allowed to perform the requested operation, and then deny it or execute it. It is important that the kernel sets the entry point when transition to kernel mode; if the application could decide the kernel entry point, a malicious application could enter the kernel at a point

where the validation of arguments etc. is skipped.

2.3 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.

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.

To reduce the risk of mistakes in the kernel, OS designers can make the lines of code that run in kernel mode small. Most of the operating system doesn't need access to privileged instructions, and can thus run as ordinary user-level applications, with which applications interact with through messages. This kernel organization is called a *microkernel*.

In a microkernel, the kernel interface consists of a few low-level functions for starting applications, performing I/O, sending messages to applications, etc. This organization allows the kernel to be implemented with a few lines of code, since it doesn't do much, as most functionality of the operating system is implemented by user-level servers.

In the real-world, one can find both monolithic kernels and microkernels. For example, Linux is mostly implemented as a monolithic kernel, although some OS functions run as user-level servers (e.g., the windowing system). Xv6 is implemented as a monolithic kernel, following most Unix operating systems. Thus, in xv6, the kernel interface corresponds to the operating system interface, and the kernel implements the complete operating system. Since xv6 doesn't provide many functions, its kernel is smaller than some microkernels. xv6' would inherit the monolithic kernel design from original xv6 implementation.

2.4 Process overview

The unit of isolation in xv6 (as in other Unix operating systems) is a *process*. The process abstraction prevents one process from wrecking or spying on another process' memory, CPU, file descriptors, etc. It also prevents a process from wrecking the kernel itself (i.e., from preventing the kernel to enforce isolation). The kernel must implement the process abstraction with care because a buggy or malicious application may trick the kernel or hardware in doing something bad (e.g., circumventing enforced isolation). The mechanisms used by the kernel to implement processes include user/kernel mode flag, address spaces, and time slicing of threads, which this subsection provides an overview of.

To be able to enforce isolation, a process is an abstraction that provides the illusion to a program that it has its own abstract 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 x86 page table translates (or "maps") a *virtual address* (the address that an x86 instruction manipulates) to a *physical address* (an address that the processor chip sends to main memory).

In xv6', since the memory management hardware merely consists of a TLB in MIPS architecture, most part of virtual-to-physical translation is done *manually*, that is, by software. Memory management would be discussed in detail in chapter 3.

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 (for malloc) that the process can expand as needed.

Each process's address space maps the kernel's instructions and data as well as the user program's memory. When a process invokes a system call, the system call executes in the kernel mappings of the process's address space. This arrangement exists so that the kernel's system call code can directly refer to user memory. In order to leave room for user memory to grow, xv6's address spaces map the kernel at high addresses, starting at 0x80100000. xv6', running on MIPS, maps the kernel in a segment called `CKSEG0` , which corresponds to the lower 512MB of physical address. `arch/mips/include/asm/addrspace.h:115`

The xv6 kernel maintains many pieces of state for each process, which it gathers into a `struct proc`. xv6' treats process and threads alike as *tasks*, a concept borrowed from Linux, and the state information is stored inside a `task` structure. `include/sched/task.h:39` A task's most important pieces of kernel state are its page table, its kernel stack, and its run state. We'll use the notation t-¿xxx (or p-¿xxx for compatibility with original xv6) to refer to elements of the process/task structure.

Each process has a thread of execution (or *thread* for short) that executes the process's instructions. A thread can be suspended and later resumed. To

switch transparently between processes, the kernel suspends the currently running thread and resumes another process's thread. Much of the state of a thread (local variables, function call return addresses) is stored on the thread's stacks. Each process has two stacks: a user stack and a kernel stack (`p->kstack`). 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 (for a system call or interrupt), 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. A process's thread alternates between actively using its user stack and its kernel stack. 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.

If time permitting, xv6' would support processes with multiple threads. This is not my major goal since threads and processes don't differ very much in design of Linux, and thus, xv6'.

When a process makes a system call, the processor switches to the kernel stack, raises the hardware privilege level, and starts executing the kernel instructions that implement the system call. When the system call completes, the kernel returns to user space: the hardware lowers its privilege level, switches back to the user stack, and resumes executing user instructions just after the system call instruction. A process's thread can "block" in the kernel to wait for I/O, and resume where it left off when the I/O has finished.

In original xv6, `p->state` indicates whether the process is allocated, ready to run, running, waiting for I/O, or exiting. This is slightly different in xv6', where state of a process is indicated by `t->state` and `t->flags` together.

In the old implementation `p->pgdir` holds the process's page table, in the format that the x86 hardware expects. xv6 causes the paging hardware to use a process's `p->pgdir` when executing that process. A process's page table also serves as the record of the addresses of the physical pages allocated to store the process's memory. In xv6' the page table is stored inside `t->mm`, a pointer pointing to a memory mapping structure, which is split into two parts, one hardware-dependent and the other hardware-independent. The hardware-related part is stored in an inner structure `arch_mm_t`. In current implementation, `arch_mm_t` merely holds the address of page global directory of the process. This structure organization is equivalent to that of original implementation, with the advantage of decoupling hardware-specific page table and hardware-irrelevant memory management code.

`include/mm/vmm.h:86`
`arch/mips/include/asm/mm/hier/vmm.h:20`

2.5 Code: the first address space

To make the xv6 organization more concrete, we look how the kernel creates the first address space (for itself), how the kernel creates and starts the first process, and the first system call that the first process makes. By tracing these operations we see in detail how xv6 provides strong isolation for processes. The first step in providing strong isolation is setting up the kernel to run in its own address space.

When it comes to Loongson Multitech Board, the machine bootstraps itself by loading program from a built-in PMON ROM which is already provided by the manufacturer. PMON downloads the operating system kernel remotely via TFTP after hardware initialization, and automatically detects the entry point and jumps there. The kernel must reside in `CKSEG0`, a segment inaccessible from other modes, where virtual addresses are linearly mapped to physical address by taking the lower 29 bits, not passing through TLB during translation. By convention the lower 256MB to 512MB actually corresponds to I/O devices or ROM, so the kernel can only take the lower 0-256MB address. The phenomenon one can observe is that xv6' loads itself in virtual address `0xffff ffff 8030 0000`.

`arch/mips/include/asm/addrspace.h:115`

In MIPS64 architecture, the virtual address layout is fixed, and is described in `arch/mips/include/asm/addrspace.h`. By enforcing the virtual addresses directly used by kernel to reside in `XKPHY`, a kernel-mode-only address space, where virtual addresses are mapped to physical ones in a manner similar to that in `CKSEG0`, it is no longer necessary to make separate virtual address mapping for kernel, and to switch page table during user/kernel mode transition. However, the address space `XKPHY` is MIPS64-specific. In 32-bit MIPS architecture a different mechanism should be developed.

`arch/mips/include asm/addrspace.h:92`

The kernel entry points to a small piece of assembly code, which is almost always architecture-specific. In xv6', all the assembly snippet would do is initializing various co-processor registers, boot arguments, initial kernel stack and processor information, and then transfer control to C code.

`arch/mips/entry.S`

`kern/init.c`

2.6 Code: creating the first process

Now the kernel runs within its own address space, we look at how the kernel creates user-level processes and ensures strong isolation between the kernel and user-level processes, and between processes themselves.

The procedure which spawns initial tasks is `task_init`. `task_init` spawns two tasks named `idle` (`idle_init`) and `init` (`initproc_init`). `initproc_init` dynamically allocates a `task` structure first by calling `task_new`, which is called on allocating every new task. The initializer then allocates and sets up a new memory mapping object, creating a new page table on the way (`task_setup_mm`).

`kern/sched/task.c:182`
`kern/sched/task.c:140`
`kern/sched/task.c:88`
`kern/sched/task.c:26`
`kern/sched/task.c:48`

The initializer should then set up the process context and the kernel stack. `task_setup_kstack` tries to allocate a kernel stack in kernel space first, and then preserves a portion on the top for storing process context for context switches. The procedure for setting up initial process context is architecture-specific as shown in `task_bootstrap_context` . In short, this routine would make the process return to the entry of C function `forkret` , which would in turn transfer control to function `arch_forkret` written in assembly, which would restore a previously built trap frame for restoring and returning in a way similar to that of finishing handling an exception. This setup is mostly the same for ordinary `fork`s.

`kern/sched/task.c:26`

`arch/mips/sched/task.c:41`
`arch/mips/syscall/fork.c:26`
`arch/mips/syscall/forkret.S`

Now the initializer should build a trap frame inside the kernel stack in a manner so as to make the process have the illusion that it is returning from a trap handler after a context switch. Building such trap frame is architecture-dependent, and is accomplished by

1. `task_init_trapframe` (for filling some process-common trap frame entries), — arch/mips/sched/task.c:21
2. `set_task_user` (for setting the task to run in user mode), — arch/mips/sched/task.c:53
3. `set_task_enable_intr` (to enable interrupt), — arch/mips/sched/task.c:62
4. `set_task_startsp` (for setting up stack pointer register to point to top of user stack, called by `set_task_ustack` for user stack setup), — arch/mips/sched/task.c:68, kern/sched/task.c:71
5. `set_task_main_args` (for passing arguments of `main()`), and — arch/mips/sched/task.c:107
6. `set_task_entry` (for specifying program entry) together. — arch/mips/sched/task.c:115

After execution of `set_task_main_args`, arguments should be placed properly above the per-process user stack, which should reside in a separate user page. User stack allocation is done by `set_task_ustack` — kern/sched/task.c:71

Unlike original xv6, the first process is loaded from a separate ELF program, named `init`, preferably stored on the disk, but now embedded in the kernel binary file as the box currently lacks external storage. `init` would become the xv6' equivalent of contemporary `init` program in System V or BSD, or `systemd` in Linux, but currently it does nothing rather than emitting a bunch of lines to the serial. To actually load the program into memory and execute it, the kernel locates the embedded `init` binary file by referencing the symbol `_binary_ramdisk_init_init_start`, generated by GNU `ld` utility while linking. The procedure `task_load_elf_kmem` loads each loadable segment into memory, mapping them to virtual address specified by the ELF program headers. The creation is complete after the entry is located by looking into the corresponding field inside the ELF header. — ramdisk/init/init.c, kern/sched/task.c:113, kern/syscall/execve.c:118

The process is now ready to run, and the initializer finalizes the procedure by setting appropriate PID, process name, and state (`TASK_RUNNABLE`)

2.7 Code: running the first process

Now that the first process's state is prepared, it is time to run it. After `main` bootstrapped everything, `mpmain` calls `scheduler` to start running processes (NYI). `Scheduler` (NYI) looks for a process with `p->state` set to `TASK_RUNNABLE`, and theres only one: `init`. It sets the per-cpu variable `current_task` to the process it found and switches page table during context switch in `switch_context` , by exploiting and changing the ASID field in `CP0_ENTRYHI` register . switchuvm also sets up a task state segment SEG_TSS that instructs the hardware to execute system calls and inter- rupts on the processs kernel stack. We will re-examine the task state segment in Chapter 3. — arch/mips/include/asm/thread_info.h:47, arch/mips/sched/switch.S