

Discussion Section

Week 3

Working on real Operating Systems

- Very Large Codebases
 - The Linux Kernel has 30.34 million lines of code!
- Very complex Codebases
 - Operating Systems need to be resilient
 - Define how software interacts with hardware
 - Orchestrates how programs interact with other programs and systems
- We start small

v6 Manual

A COMMENTARY ON THE SIXTH EDITION UNIX OPERATING SYSTEM

12.7 **exec** (3020)

This system call, #11, changes a process **exec**uting one program into a process **exec**uting a different program. See Section “**EXEC**(II)” of the UPM. This is the longest and one of the most important system calls.

3034: “namei” (6618) (which is discussed in detail in Chapter 19) converts the first argument (which is a pointer to a character string defining the name of the new program) into

an “inode” reference. (“inodes” are essential parts of the file referencing mechanism.);

3037: Wait if the number of “**exec**”s currently under way is too large (See the comment on line 3011.);

3040: “getblk(NODEV)” results in the allocation of a 512 byte buffer from the pool of buffers. This buffer is used temporarily to store in core, that information which is currently in the user data area, and which is needed to start the new program. Note that the second argument in “u.u.arg” is a pointer to this information;

3041: “access” returns a non-zero result if the file is not **exec**utable. The second condition examines whether the file is a directory or a special character file. (It would seem that by making this test earlier, e.g. just after line 3036, the efficiency of the code could be improved.);

3052: Copy the set of arguments from the user space into the temporary buffer;

3064: If the argument string is too large to fit in the buffer, take an error exit;

3071: If the number of characters in the argument string is odd, add an extra, null character;

3090: The first four words (8 bytes) of the named file are read into “u.u.arg”. The interpretation of these words is indicated in the comment beginning on line 3076 and, more fully, in the section “A.OUT(V)” of the UPM.

Note the setting of “u.u-base”, “u.u-count”, “u.u_offset” and “u.u_segflg” preparatory to the read operation;

3095: If the text segment is not to be protected, add the text area size to the data area size, and set the former to zero;

3105: Check whether the program has a “pure” text area, but the program file has already been opened by some other program as a data file. If so, take the error exit;

3127: When this point is reached, the decision to **exec**ute the new program is irrevocable i.e. there is no longer the opportunity to return to the original program with an error flag set;

3129: “expand” here actually implies a major contraction, to the “per process data” area only;

3130: “xalloc” takes care of allocating (if necessary) and linking to the text area;

3158: The information stored in the buffer area is copied into the stack in the user data area of the new program;

3186: The locations in the kernel stack which contain copies of the “previous” values of the registers in user mode are set to zero, except for r6, the stack pointer, which was set at line 3155;

3194: Decrement the reference count for the “inode” structure;

3195: Release the temporary buffer;

3196: Wake up any other process waiting at line 3037.

xv6

- Based on v6, the first public release version of the Unix Operating System (1975)
- Created at MIT in 2005
- Runs in an emulated environment, using qemu (Quick Emulator)

Features of xv6

- Minimalist Kernel, around ~10K lines of code
- Round Robin Scheduler
- Process Creation via `fork()`, context switching
- Segmentation and Paging for Memory Management
- File System
- IPC using pipes ("`|`")
- System Calls
 - `read()`, `write()`, `open()`, `close()`, `uptime()`, `fork()` etc
- Interrupt-driven processing
- User Space utilities
 - Shell, `ls`, `cat`, `grep`, `echo`

How to build and run xv6

1. Clone this repository (Note, we are using x86 version) [GitHub - mit-pdos/xv6-public: xv6 OS](https://github.com/mit-pdos/xv6-public)
2. Cd to xv6-public directory
3. make
4. Make-qemu-nox

```
Booting from Hard Disk..xv6...
```

```
cpu0: starting 0
```

```
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
```

```
init: starting sh
```

```
$ █
```

How to debug xv6

- Check if you have `.gdbinit.tmpl` in your `xv6` directory
 - We will include this in your solution folder
 - Check using `ls -a` (`.gdbinit.tmp` is hidden)
- Modify your `gdb` config (`gdbinit`), found in `/home/USERNAME/.config/gdb`
 - If you cannot find this path, try `mkdir -p /home/USERNAME/.config/gdb`
 - Add a file `gdbinit` in the `gdb` directory
- You need to add this line to the top of `/home/USERNAME/.config/gdb/gdbinit`
 - `add-auto-load-safe-path [absolute path to xv6 directory]/.gdbinit`
- Navigate back to your `xv6` directory
- Run `make qemu-nox-gdb`
- Open another `ssh` connection, or use `tmux` to create another panel
- Navigate to `xv6` directory and run `gdb`
- Run `gdb` command `continue` and you should see the `xv6` run in the other panel

You can also use print statements, `printf()` works in userspace, and `cprintf` will work in kernel space.

How to add and run a user program

- Create your program (in this case, let's call it `usertest.c`)
- Open Makefile
- Add `_usertest` to `UPROGS` (Line 168)
- Add `usertest.c` to `EXTRA` (Line 251)

Compile and run xv6

You should be able to launch the user program from the xv6 prompt

Debugging a user Program

- Let's say you have a stub `_test` that you added in `USERPROGS`, which reflects some executable "test"
- In `gdb`, type
 - `add-symbol-file _test`
- This will load up the symbol table (i.e., the debugging information) for `test`.
- In `gdb`, type
 - `break test.c:[line number]` to set a breakpoint
- Hit `continue`, switch to `xv6` shell, run `./test`, and it should break at that line
- Try using `layout src` to get a full view of code, and debugging information

Practical Demo!

Let's take a look at

1. How to compile and run
2. How to debug
3. How to add user programs
4. Look at internals?