xv6: Implementing Authentication, Access Control, and ASLR

This document is based on commit 077323a.

Foreword

When in doubt, first look in the appendix, then use a search engine.

Authentication

Prerequisites

General knowledge of the C programming language and Linux system calls. Important knowledge includes:

- Header files and the `#include` directive.
- The concepts of declaration and definition.
- Types.
- Memory management, including allocation and deallocation.
- Pointers.
- Structures.
- File I/O conventions, including the file descriptor and the system calls `open`, `read`, `write`, and `close`.
- The `exec` and `fork` system calls.

Familiarity with the `man` program.
Knowledge of what Unix, Linux, and POSIX are.

Background

Security decisions can be based around a user identity. This allows the design of security policies that follow the principle of least privilege. For example, a normal user on a system will be given only the minimum set of privileges needed to perform usual tasks, while only the superuser is given all privileges. Thus, it is very important for a system to correctly determine the identity of an entity. The process of determining the identity of a user is called authentication. Historically, authentication has most often been implemented through the use of passwords. Other forms of authentication include multi-factor authentication and biometric authentication.

Goal

User authentication will be implemented into xv6. The implementation will conform to existing operating systems standards. xv6 will prompt for login after initializing, after which, the user will be able to input commands into the shell.

1. Create user account abstraction.

User account abstraction will allow programs to create and get information for user accounts easily.

A.

POSIX provides user account abstraction with `pwd.h`. See the `man` page for `pwd.h`. To provide this abstraction to user programs, organize the functions and structures in the following sections into a new userspace library conforming to POSIX. See “Appendix”, “Creating a new userspace library function”, “Creating many functions of the same scope”.

B.

POSIX specifies that user account information is stored in a `passwd` structure. This structure can be manipulated by a program at runtime. However, the structure does not exist outside of the lifespan of a process. In Linux, persistent user account information is located on disk at the passwd file. For example, a passwd file in Linux may read:

```
root:x:0:0::/root:/bin/bash
bin:x:1:1::/:/usr/bin/nologin
daemon:x:2:2::/:/usr/bin/nologin
...
username:x:1000:1001::/home/username:/bin/bash
...
```

In summary, an entry may have the following format:

```
user:pass:1000:1001
```

Where:

- The first field, `user`, is the username.
- The second field, `pass`, is the password. In Linux, this field is omitted. However, this implementation will insert a hashed password and salt into this field.
- The third field, `1000`, is the user ID.
- The fourth field, `1001`, is the group ID.

This format is just one example. Other fields may be added to the passwd file entry format, but at least these fields must exist.
Implement the `passwd` structure into `user/pwd.h`.

C.

POSIX specifies functions that interact with the passwd file. Among these functions are:

- `void setpwent(void)`. This function resets the offset for the passwd file to the beginning (ie, to the first entry).
- `void endpwent(void)`. This function closes the passwd file.
- `struct passwd *getpwent(void)`. This function returns the next entry of the passwd file.
- `struct passwd *getpwnam(const char *)`. This function returns the entry of the passwd file whose username matches the given argument.
- `struct passwd *getpwuid(uid_t)[1]`. This function returns the entry of the passwd file whose user ID matches the given argument.

[1] The type defined as `uid_t` can simply be replaced with an appropriate existing type, such as `uint`, or it can be implemented.

See each function's respective `man` page. Implement at least the functions listed above.
Linux provides a `putpwent` function. This function creates a user account entry. See the `man` page for `putpwent`. Implement `putpwent`.

2. Create `useradd`

A.

In Linux, the `useradd` program creates users and the `passwd` program sets the password for users. For simplicity, we will combine the functions of these two programs into one called `useradd`.
`useradd` should:

1. Prompt for a username and take user input.
    - If the username already exists in the passwd file, the user should be notified and the program should exit.
2. Prompt for a password and take user input.
    - Ideally, the user input should not be echoed onto the terminal.
3. Hash and salt the plain-text password such that it can be safely stored onto a readable file and be read later. See section "B.".
4. Use the gathered information to create a user account entry.

Create the `useradd` program. See "Appendix", "Creating a new user program". The functions available for use in a user program are listed as function prototypes at each included header file, for example `user/user.h`.

B.

Ideally, the password should be hashed with a cryptographic hash function and salted with a large (32 or 64 bit) number. For simplicity, in this implementation, the password will be hashed with the Jenkins `one_at_a_time` hash function (which is not cryptographic) and salted with a 32 bit number:

1. Create a library function that returns a random 32 bit number. The `uptime` and `getpid` system calls can be used to provide seed numbers. The return type can be `unsigned int` or `uint`. See "Appendix", "Creating a new userspace library function", "Creating a few functions".
2. Create a library function that implements the Jenkins `one_at_a_time` hash function. The code is provided below.

```c
uint jenkins_one_at_a_time_hash(const char* key, uint length) {
  uint i = 0;
  uint hash = 0;
  while (i != length) {
    hash += key[i++];
    hash += hash << 10;
    hash ^= hash >> 6;
  }
  hash += hash << 3;
  hash ^= hash >> 11;
  hash += hash << 15;
  return hash;
}
```

See “Appendix”, “Creating a new userspace library function”, “Creating a few functions”.

3. Use the functions above to create a salted hash of the plain-text password. Concatenate the plain-text password with a random 32 bit integer and hash the resulting string.
4. The salted hash and salt itself is stored in a user account entry as the password field. A ‘$’ character can be used to separate the hash from the salt.

3. Implement process identities

In Unix, each process is owned by a user and a group. The identity of a process is marked by each process’s user ID (UID) and group ID (GID). This mechanism allows for the enforcement of policies which limit the scope of a process’s actions depending on which user or group owns that process.
To store per-process user ID and group ID information, the `struct proc` at `kernel/proc.h` should be modified. See “Appendix”, “Modifying the proc struct”, “Creating an inheritable field”. Without any initialization code, these new fields will be initialized to zero. This is desired behavior. The root user conventionally has the UID of zero and the GID of zero.
POSIX provides the following functions to get and set the real UID and GID of a process:

- `uid_t getuid(void)`
- `gid_t getgid(void)`
- `int setuid(uid_t uid)`
- `int setgid(gid_t gid)`

Implement these functions as system calls. See “Appendix”, “Modifying the kernel”, “Creating a new system call”.

4. Create `whoami`

In Linux, the `whoami` command outputs information about the current user.
`whoami` should:

1. Output the name of the current user.

Create the `whoami` program. See “Appendix”, “Creating a new user program”.

5. Create `login`

In Linux, the `login` command is called by the first process, the init process, in order to authenticate the user. After successful authentication, the shell is started and then the user is able to input commands.

`login` should:

1. Check if the passwd file exists.
    - If the file does not exist, then the root user should be created.
2. Prompt for a username and take user input.
3. Prompt for a password and take user input.
    - Ideally, the user input should not be echoed onto the terminal.
4. Check if a user account exists matching the provided login.
    - If none exist, then login information should be requested again. The user should not be allowed to retry often.
    - To check if the given password matches the recorded credentials, concatenate the given password with the recorded salt and hash the result. If the hash matches the recorded hash, then the given password matches the recorded credentials.
5. Set the UID and GID of the process to be that of the user.
6. Execute the shell.

See the `man` page for `login`.

Create the `login` program. See “Appendix”, “Creating a new user program”.

6. Modify the init process

Modify the `init` program at `user/init.c` to start the `login` program instead of the shell, `sh`.

Access control

Prerequisites

General knowledge of the C programming language and Linux system calls. Important knowledge includes:

- Header files and the `#include` directive.
- The concepts of declaration and definition.
- Types.
- Memory management, including allocation and deallocation.
- Pointers.
- Structures.
- File I/O conventions, including the file descriptor and the system calls `open`, `read`, `write`, and `close`.
- The `exec` and `fork` system calls.

Familiarity with the `man` program.
Knowledge of what Unix, Linux, and POSIX are.

Background

Files in a system are commonly shared between the system's users. Usually, it is desired that security policies are put in place to determine what files a user can access. For example, it may be desired that some personal files are made private to one user, and that other files are made available only to a select group of users, and that other files are made available to every user in the system. In Unix, access control was implemented with access control lists (ACLs). Specifically, every file was associated with permission bits. These permission bits control which users or groups can access a file and what they are allowed to do to the file.

Goal

Unix-style access control will be implemented into xv6.

1. Implement process identities

In Unix, each process is owned by a user and a group. The identity of a process is marked by each process's user ID (UID) and group ID (GID). This mechanism allows for the enforcement of policies which limit the scope of a process's actions depending on which user or group owns that process.
To store per-process user ID and group ID information, the `struct proc` at `kernel/proc.h` should be modified. See "Appendix", "Modifying the proc struct", "Creating an inheritable field". Without any initialization code, these new fields will be initialized to zero. This is desired behavior. The root user conventionally has the UID of zero and the GID of zero.
POSIX provides the following functions to get and set the real UID and GID of a process:

- `uid_t getuid(void)`
- `gid_t getgid(void)`
- `int setuid(uid_t uid)`
- `int setgid(gid_t gid)`

Implement these functions as system calls. See “Appendix”, “Modifying the kernel”, “Creating a new system call”.

2. Implement file ownership

A.

POSIX specifies the \`stat\` function to get information about a file. In Linux, the \`stat\` system call returns information regarding which user and group owns the file as the \`st_uid\` and \`st_gid\` fields. See the \`man\` page for \`stat(2)\`.
Implement file ownership information. One way to do this is to modify the inode structures. See “Appendix”, “Modifying the dinode or inode structs”, “Creating a new field”.

B.

POSIX specifies the \`chown\` function to set file ownership. \`chown\` takes a file’s path as one of its arguments. See the \`man\` page for \`chown(3p)\`. Linux also provides the \`fchown\` system call. \`fchown\` takes a file descriptor as an argument instead of a path. See the \`man\` page for \`chown(2)\`.
Note that only the file’s owner or the root user should be able to change its ownership.
Modify the kernel to be able to set file ownership. Implement at least one of \`chown\` or \`fchown\`. See “Appendix”, “Modifying the kernel”. Note that inodes will have to be manipulated.

C.

In Linux, the \`chown\` program sets ownership for a file.
\`chown\` should:

1. Take in the desired ownership as the first argument.
2. Take in file paths as the next arguments.
3. Use a chown system call to set file ownership.

See the \`man\` page for \`chown\`.
Create the \`chown\` program. See “Appendix”, “Creating a new user program”. See the source code of other user programs, such as \`user/mkdir.c\`, \`user/rm.c\`, or \`user/cat.c\` for examples on how to write user programs.

3. Implement access control lists

A.

In Unix, the ACL is implemented in each file’s inode, along with the file’s type. This ACL, called the permission bits or permission mode, is 12 bits in size. Usually, the permission bits are formatted in octal or conforming to the regular expression `[rwx-]{9}`. For example, the Linux `stat` command may output:

- `Access: (0644/-rw-r--r--)`

Where:

- `0644` is the octal representation of the permission bits.
- `rw-r--r--` is the human readable representation of the last nine permission bits.
  - The first group of bits, called “owner”, apply to the file’s owner.
  - `rw-`: the owner is allowed to read from and write to the file, but not execute the file.
  - The second group of bits, called “group”, apply to the group that owns the file.
  - `r--`: users in the group are allowed to read from the file, but not write to or execute the file.
  - The third group of bits, called “other”, apply to every other user in the system.
  - `r--`: others are allowed to read from the file, but not write to or execute the file.

In Linux, the permission bits for a file are accessible through the `stat` system call as the `st_mode` field. See the `man` page for `stat(2)` and the `man` page for `inode`.
Implement access control lists. One way to do this is to modify the inode structures. See “Appendix”, “Modifying the dinode or inode structs”, “Creating a new field”.

B.

POSIX specifies the `chmod` function to change the permission mode of a file. `chmod` takes a file’s path as one of its arguments. See the `man` page for `chmod(3p)`. Linux also provides the `fchmod` system call. `fchmod` takes a file descriptor as an argument instead of a file path. See the `man` page for `chmod(2)`.
Note that only the file’s owner or the root user should be able to change its permission mode.
Modify the kernel to be able to set a file’s permission mode. Implement at least one of `chmod` or `fchmod`. See “Appendix”, “Modifying the kernel”. Note that inodes will have to be manipulated.

C.

In Linux, the `chmod` program sets a file’s permission mode.
`chmod` should:

1. Take in the desired permission mode as the first argument.
2. Take in file paths as the next arguments.
3. Use a chmod system call to set the permission mode.

See the `man` page for `chmod`.

Create the `chmod` program. See “Appendix”, “Creating a new user program”. See the source code of other user programs, such as `user/mkdir.c`, `user/rm.c`, or `user/cat.c` for examples on how to write user programs.

4. Implement access authorization

A.

In Unix v6, the `access` kernel function controls read, write, and execution authorization. POSIX.1-2017 specifies the `access` function to check accessibility for a file. This function checks if a file is either readable, writable, or executable. Specifically, `access` ensures that:

- The root user always has access.
- A file is readable if:
  - the read bit for others is set, or
  - the read bit for the group is set and the calling process is owned by the group that owns the file, or
  - the read bit for the owner is set and the calling process is owned by the user that owns the file.
- A file is writable if:
  - the write bit for others is set, or
  - the write bit for the group is set and the calling process is owned by the group that owns the file, or
  - the write bit for the owner is set and the calling process is owned by the user that owns the file.
- A file is executable if:
  - the execute bit for others is set, or
  - the execute bit for the group is set and the calling process is owned by the group that owns the file, or
  - the execute bit for the owner is set and the calling process is owned by the user that owns the file.

Create this `access` kernel function. Other kernel functions can be created for internal use. See “Appendix”, “Modifying the kernel”.

B.

In Linux, processes are allowed the following actions, depending on the permissions set on a file:

- If a file is readable, then its contents can be read and copied. Otherwise not.
- If a file is writable, then its contents can be modified. Otherwise not.
- If a file is executable, then it can be executed. Otherwise not.
- If a directory is readable, then the files within it can be listed and copied to other directories. Otherwise not.
- If a directory is writable and executable, then files can be added or removed from within it. Otherwise not.

- If a directory is executable, then it can be entered (ie, the current directory can be changed to that directory). Otherwise not.

Modify kernel functions to implement access authorization. At least the following existing kernel functions at `kernel/sysfile.c` will be modified in the implementation of access authorization:

- `sys_open`
- `create`
- `sys_exec`
- `sys_link`
- `sys_unlink`
- `sys_chdir`

5. Implement default file creation ownership and mode

A.

In Linux, newly created files are owned by the identity associated with the calling process. Modify the `create` kernel function at `kernel/sysfile.c:create` to implement this process-to-file identity inheritance. Note that inodes will have to be manipulated. Although a new kernel function will not be written, see “Appendix”, “Creating a new kernel function”, “Writing a kernel function that manipulates inodes” to better understand how `create` is written. The kernel function at `kernel/proc.c:myproc` should be called to retrieve process information.

B.

In Linux, system calls that create new files require a `mode` argument from which a file’s permission mode is set. For example, `open` is declared as `int open(const char *pathname, int flags, mode_t mode);`, and `mkdir` is declared as `int mkdir(const char *pathname, mode_t mode);`. Ideally, every such system call should be modified to take in a `mode` argument. However, for simplicity, a simpler feature can be implemented instead. Modify the `create` kernel function at `kernel/sysfile.c:create` to set a default permission mode for newly created files. See section “A”.

6. Implement set-user-ID capabilities

In Unix, a process can set its user ID only if it has the capability to. This capability can be granted if the executable file the process originates from has the set-user-ID bit (the first or most significant bit) of its permission bits set. Since processes can gain privileges by setting its UID to some other value, like the root UID, this mechanism ensures that only processes with the appropriate privileges can gain them. For example, without this mechanism, any user process could set its UID to zero and gain root privileges, allowing the process full control of the system. Implement set-user-ID capabilities. One way to do so is to modify `kernel/exec.c` to grant a process set-UID capabilities if the file has the set-UID bit set. Modify the `setuid` and `setgid` kernel functions by writing proper authorization mechanisms.

Address space layout randomization (ASLR)

Prerequisites

General knowledge of the C programming language and Linux system calls. Important knowledge includes:

- Header files and the `#include` directive.
- The concepts of declaration and definition.
- Types.
- Memory management, including allocation and deallocation.
- Pointers.
- Structures.
- File I/O conventions, including the file descriptor and the system calls `open`, `read`, `write`, and `close`.
- The `exec` and `fork` system calls.

Familiarity with virtual memory.

Background

Physical memory is virtualized by the machine. Each user process is given its own virtual address space and thus prevents the process from accessing the memory held by another process. However, interaction between the user and process, combined with the fact that instructions are stored in memory allows for a user to perform arbitrary code execution.
A variety of attacks take advantage of these vulnerabilities, such as buffer overflow attacks. These attacks usually require the memory addresses of specific data to be known.
To defend against these attacks, various systems have been developed. One such system is address space layout randomization (ASLR). ASLR randomizes the starting address of various memory segments in a process's address space. With ASLR, an attack that uses an assumed memory location will be far less likely to succeed.

Goal

Address space layout randomization will be implemented into xv6. Compared to unmodified xv6, this implementation will be effective in defending against buffer overflow attacks.

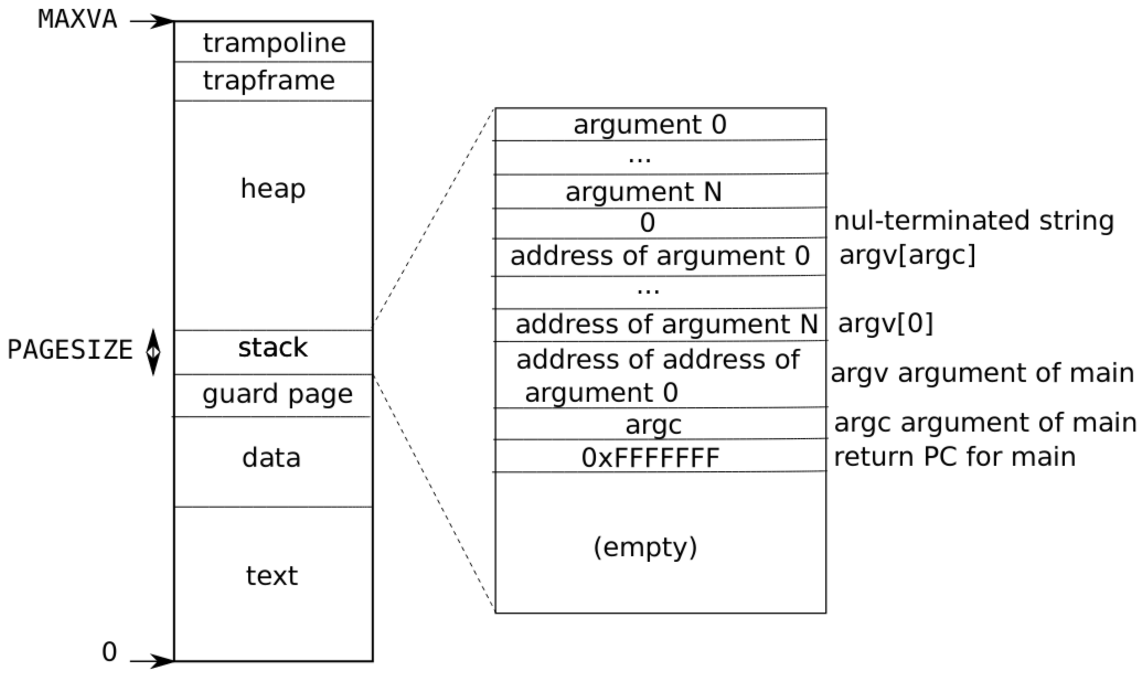
1. Create random number generator

The random number generator (RNG) will create the value of the offsets for each memory segment. An ideal RNG would be fast and have high entropy. This ideal RNG would keep performance overhead minimal while greatly reducing the chance that an attack will succeed.
Create a random number generator. The RNG code can be placed in its own file (such as at `kernel/random.c`) or can be placed at the file where it is used (at `kernel/exec.c`). The kernel's

trap function at `kernel/trap.c:kerneltrap` can be used to provide a source of randomness. This function is called whenever the operating system traps into kernel code.

2. Modify the executable file loader

xv6 userland programs use the `exec` system call to load executables. The kernel uses the program at `kernel/exec.c` to load executables.
xv6 follows the Executable and Linking Format (ELF) when loading executable files. The function at `kernel/exec.c:exec` reads the ELF header of the executable file, then loads the program segment (also called, text segment or code segment) into process memory. ELF allows for multiple program segments to be loaded into memory, but xv6 loads only one.
After loading the program segment, `kernel/exec.c:exec` initializes the process's stack, does some other things, then sets the initial program counter to the main function (ie, the entry address specified by the ELF header).
There are two memory segments whose offsets can be easily modified: the program segment and the stack segment. The location of the two segments in the user address space is depicted below.



A. Randomize the program segment offset

The program segment is loaded into process memory in the following way. While reading this list, find the code at `kernel/exec.c:exec`.

1. User virtual memory is allocated to accommodate for the program segment with the function `uvmalloc`.

2. The program segment is loaded onto the allocated memory with the function `kernel/exec.c:loadseg`.

There can be several approaches used to modify the program segment offset. One approach is to create a random offset once when the machine boots and to use that offset for every executable loaded after. Another approach is to create a random offset every time an executable is loaded. Both approaches are simple. The latter approach may incur slightly more overhead than the first approach, but neutralizes the danger of one memory address leak whereas the first approach would be compromised.

In any case, randomize the program segment offset. To do this, create a random value to use as an offset. The value should be page aligned (ie, a multiple of the `PGSIZE` constant). Add this value onto the last argument of the `uvmalloc` function call written at the “load program into memory” section. Also, add this same value onto the last argument of the `loadseg` function call written after, and to the initial program counter, `p->trapframe->epc`, which is modified at the end of `kernel/exec.c:exec`.

B. Randomize the stack segment offset

After the program segment is loaded, the stack is initialized. The stack segment is initialized in the following way. While reading this list, find the code at `kernel/exec.c:exec`.

1. Two pages are allocated for the stack.
2. Those two pages are cleared.
3. The stack pointer, `sp`, is placed at the end of those two pages.
4. The address of the end of the stack, `stackbase`, is set to one page before the stack pointer.

xv6 allocates just one page for a process’s stack. The page before the stack acts as a guard page. If a process tries to write to the guard page, the machine will generate an exception and likely kill the offending process.

Randomize the stack segment offset. To do this, create a random value to use as an offset. The value should be page aligned (ie, a multiple of the `PGSIZE` constant). Replace the current value of `2*PGSIZE` with this random value.

3. Implement configuration

In situations where ASLR is not wanted, the feature should be able to be configured off. Implement ASLR configuration. Configuration can be implemented in various ways. A variable could be used to globally configure ASLR during xv6’s compile time. In Linux, ASLR can be globally configured by interacting with the `/proc/sys/kernel/randomize_va_space` file. Linux also includes per-process ASLR configuration with the `personality` system call.

Appendix

Setting up xv6

Although any Linux machine can be used, Ubuntu 20.04 has been tested to work well with xv6 and xv6 development. If a machine running Ubuntu is not available, the use of Docker is recommended to create an Ubuntu environment. See the repository at https://github.com/jansenmtan/xv6-docker to set up a Docker container made for xv6 development.

To set up xv6:

1. Get xv6 from the repository at https://github.com/mit-pdos/xv6-riscv.
2. Follow the “BUILDING AND RUNNING XV6” section at the README.
    a. Ubuntu provides the \`gcc-riscv64-linux-gnu\` package, which provides the toolchain mentioned in the README.
    b. Ubuntu provides the \`qemu\` and \`qemu-system-misc\` packages, which provide the build of QEMU mentioned in the README.

Ensure that QEMU’s version is less than 6.0.0. Recent versions of QEMU will not boot xv6. QEMU version 4.2.1 has been tested to work.

Debugging xv6: avoiding pain and tedium

Debugging environment

- Install the multiarch version of gdb, usually named “gdb-multiarch”. Preferably do so with your package manager.
- Consider using gdb’s TUI mode. I prefer \`layout split\`, \`layout src\`, and \`focus cmd\` when debugging. See the document at http://davis.lbl.gov/Manuals/GDB/gdb_21.html for details.
- Modify the \`-ggdb\` flag in the \`CFLAGS\` variable at the Makefile to at least \`-ggdb2\`. Doing so reveals more information while debugging with gdb. Changing the flag to \`-ggdb3\` or above can work, but is likely to cause errors.
- Try not to change the size of the window that gdb is running in. gdb likes to crash when its screen changes in size.

Debugging workflow

The debugging workflow is usually:

1. In a terminal in the xv6 directory, run \`make clean\` then \`make qemu-gdb -j 4\`.
2. If gdb is:
    - not running in another terminal window, then, in the xv6 directory, run \`gdb-multiarch\`.
    - already running in another terminal window, then type the \`target remote tcp::25000\` command to gdb.

3. Load whichever executable is being debugged with the `file` gdb command.
    - For example, if the function at `user/ls.c:ls` is being debugged, then the command to send to gdb would be `file user/_ls`.
    - If any kernel function is being debugged, then send the command `file kernel/kernel`.
4. Breakpoints can be set at this moment. See “Setting breakpoints”.
5. Continue qemu’s emulation of xv6 by sending the `continue` or `c` command to gdb.
6. Interact with xv6 to reproduce the behavior being studied. Usually breakpoints are set to pause the machine when the behavior is happening.
7. Control execution and inspect the machine state to study the program behavior.
    - Execution can be controlled by the `next`, `step`, `finish`, and `continue` commands. See “Controlling execution”.
    - The machine state can be inspected through the `info`, `print`, and `backtrace` commands. See “Inspecting the machine state”.
8. When looking to stop or restart the machine, send the Control-a+x keystrokes to QEMU. QEMU will terminate. Do not exit gdb, unless there is a reason to.

Setting breakpoints

Breakpoints are set to allow the study of program behavior.

To set a breakpoint, use the `break` or `b` gdb command. For example:
- `b 50` sets a breakpoint at line 50 of the currently loaded file.
- `b ls.c:50` sets a breakpoint at line 50 of `ls.c`, if `ls.c` is loaded.
- `b ls.c:main` or `b main` sets a breakpoint at the `main` function of `ls.c`, given that `ls.c` is loaded.

Listing currently set breakpoints is done with the `info breakpoints` or `i b` command.

Breakpoints can only be set when machine execution is paused. When execution is paused, gdb prints the “(gdb)” prompt. See “Controlling execution”.

Managing breakpoints

Sometimes a breakpoint will be hit too often. There are a few solutions to this problem.

Setting conditions on breakpoints is one solution. For example:
- `b main if argc == 2` sets a breakpoint that is only hit when the `argc` variable has a value of 2.
- `condition 2 argc == 3` adds a condition to breakpoint 2.
- `condition 15 $_streq(argv[0], “ls”)` adds a condition to breakpoint 15 so that it only hits when the `argv[0]` string matches “ls”. This condition uses the gdb convenience function `$_streq()`.

Disabling breakpoints is another solution. Disabling is done with `disable` or `dis`. Enabling breakpoints is done with `enable` or `en`. For example, `dis 5` disables breakpoint 5, while `en 5` reenables breakpoint 5.

Deleting breakpoints is also possible. Deleting is done with `delete` or `d`. For example, `d 6` deletes breakpoint 6.

Controlling execution

When a breakpoint is hit or execution is paused, execution can be controlled.

`next` or `n` executes the next line in the currently loaded file.
`step` or `s` executes the next line and if a function is called, execution will jump into that function.
`finish` or `fin` continues execution until the current function returns.
`continue` or `c` continues execution.

Sending the Control-c keystroke to gdb pauses execution at any moment. This keybinding is useful when a breakpoint is being hit too often.

Inspecting the machine state

When a breakpoint is hit or execution is paused, the values of variables or memory addresses can be inspected.

Local variables can be printed with `info locals` or `i lo`.
Variables or expressions can be printed with `print` or `p`. For example:

- `p argc` outputs the value of `argc`.
- `p 1+1` outputs 2.
- `p (int)0x80000000` outputs the value of the data located at memory address `0x80000000` as an `int`.
- `p ip` outputs the address of `ip`, given that `ip` is a `struct inode *`.
- `p *ip` outputs the fields of `ip`, given that `ip` is a `struct inode *`.
- `p ip->inum` outputs the value of the `inum` field of `ip`, given that `ip` is a `struct inode *`.

The function call order can be printed with `backtrace`.

Creating a new user program

When creating a new program at the `user/` directory, also:

- Write a new line to the `UPROGS` variable at the Makefile. The format of the new line should follow that of the others.

Lastly, if after adding a user program to Makefile, the program at `mkfs/mkfs.c` gives errors, either:

- delete a line in the `UPROGS` variable at the Makefile that specifies some other user program, or
- increase the FSSIZE constant at `kernel/param.h`.

Creating a new userspace library function

Creating a few functions

When creating a new userspace library function, the best location to write the function to is at `user/ulib.c`. In this case, also:

- Add the function prototype to `user/user.h`.

The new function will be able to be accessed by any program that includes `user/user.h`.

Creating many functions of the same scope

If many functions of the same scope are to be written, then it is better to create a new C file at `user/`. This way, they will be organized under a namespace apart from `user/user.h`.
In the case that a new C file is to be used as a userspace library:

- Append the file to the `ULIB` variable at the Makefile. For example, `$U/pwd.o` would be appended if `user/pwd.c` was created.
- Create a corresponding header file at `user/` with the same base name as the C file. For example, `user/pwd.h` would be created with `user/pwd.c`.
- In the header file, write function prototypes for functions in the C file.

The new functions will be able to be accessed by any program that includes the new header file.

Modifying the kernel

Creating a new system call

When creating a new system call under the `kernel/` directory that is to be exposed to userspace programs at `user/`, also:

- Modify`kernel/syscall.h` by defining a new constant for the new system call.
- Modify `kernel/syscall.c`:
    - Write a declaration using the `extern` keyword for the new system call.
    - Append the `syscalls` static void function pointer array (declared as `static uint64 (*syscalls[])(void)`) with an entry for the new system call.
- Modify `user/user.h` by writing a function prototype for the new system call.
- Append `user/usys.pl` with an entry for the new system call.

A system call almost always calls a kernel function. See “Creating a new kernel function”.

Writing a new system call

System calls are named with the prefix `sys_`. System calls are written with a void argument and with the return type `uint64`.
The body of a system call generally has three parts:

1. The declaration of variables used throughout the body.
2. The retrieval of argument values.
3. The main part of the body.

The retrieval of argument values is done with the functions `argint`, `argstr`, `argaddr`, and `argfd`.
For example:

- In userspace, the `fstat` system call takes in two arguments. The first is an `int fd`, and the second is a `struct stat *`. `fstat` returns an `int`.
  In the kernel, the system call would be declared as a `uint64 sys_fstat(void)`. Among the declared variables would be an `struct file *f` and a `uint64 st`. These variables would store the argument values retrieved from the statements `argfd(0, 0, &f)` and `argaddr(1, &st)`.
- In userspace, the `open` system call takes in two arguments. The first is a `char *path`, and the second is an `int omode`. `open` returns an `int`.
  In the kernel, the system call would be declared as a `uint64 sys_open(void)`. Among the declared variables would be a `char path[MAXPATH]` and an `int omode`. These variables would store the argument values retrieved from the statements `argstr(0, path, MAXPATH)` and `argint(1, &omode)`.

Creating a new kernel function

When creating a new kernel function that is not in a `kernel/sys*.c` file and it is to be used by other kernel functions and system calls, also:

- Modify `kernel/defs.h` by writing a function prototype for the new kernel function.

If the new kernel function is in a `kernel/sys*.c` file (eg, `kernel/sysfile.c`), then the kernel function should be declared as a static function.

Writing a kernel function that manipulates inodes

When writing a kernel function that manipulates inodes, follow this sequence:

1. Call `iget(uint dev, uint inum)` or `namei(char *path)` to get an inode pointer.
   - Only call `iget` if an inode needs to be referenced and allocated.
   - If an inode already exists in a `struct file`, this step must be skipped.
2. Call `ilock(struct inode *ip)`.
   - This function allows the inode's fields to be manipulated.
3. Manipulate the inode's fields.
4. Call `iunlock(struct inode *ip)`.
   - After the inode is sufficiently manipulated, it should be unlocked.
5. Call `iput(struct inode *ip)`.

Modifying the dinode or inode structs

Creating a new field

When creating a new field at either the `struct dinode`at `kernel/fs.h` or the `struct inode` at `kernel/file.h`, also:

- Modify the functions at `kernel/fs.c:iupdate` and at `kernel/fs.c:ilock` to be able to handle the new field.
- Remove the assertion that `(BSIZE % sizeof(struct dinode)) == 0` at `mkfs/mkfs.c`.

The “stat” abstraction can also be updated to include information about the new field. To update the “stat” abstraction:

- Modify the `struct stat` at `kernel/stat.h` by including the new field.
- Modify the function at `kernel/fs.c:stati` by handling the new field.

Modifying the proc struct

Creating an inheritable field

If a new field is created at the process control block at `struct proc` at `kernel/proc.h`, and the field is desired to be inheritable from parent process to child process, also:

- Modify the `fork` function at `kernel/proc.c:fork` by writing code that ensures inheritance. For example, if a `uint uid` field is created, then, before the `struct proc *np` pointer is released at `release(&np->lock)`, write `np->uid = p->uid`.

Forking

Forking allows a process to create another process. The new child process uses the same program code as the parent process. The general control flow for forking is as follows:

```
int pid = fork();     // parent calls fork
if(pid < 0){          // fork error
      ...             // handle error
}
if(pid == 0){         // if pid is zero,    then we are the child
      ...             // code for the child to run
} else if(pid > 0){  // if pid is nonzero, then we are the parent
      ...             // code for the parent to run
}
```

Sometimes, the child will execute another program:

```
if(pid == 0){         // if pid is zero, then we are the child
      char *arg[] = {"path-of-program", ..., 0};
      exec("path-of-program", arg);
      // if exec goes well, then the child will never run past here
      ... // code if an error occurs. Usually `exit(1);`
}
```

While the child runs, the parent usually waits for the child to finish:

```
int xstatus;
if(pid > 0){
      wait(&xstatus); // wait for child, pass xstatus in case of exec error
      if(xstatus){
            ... // code to handle child execution error
      }
}
... // parent continues
```

Other tips

The `procdump` function at `kernel/proc.c:procdump` is useful for debugging purposes.