Introduction to xv6 on the RISC-V Architecture

Szymon Groszkowski 193141
Kamil Śliwiński 193740

20 December 2024

Contents

1 Introduction

1.1 Purpose of the guide

The purpose of this guide is to facilitate getting started with the xv6 system, particularly its version for the RISC-V architecture. xv6 is a minimalist operating system that serves as excellent educational material for those wishing to explore the fundamentals of operating systems. Its simplicity and clarity of source code make it an ideal choice for students.

This guide aims to save time and effort for individuals interested in experimenting with and learning about xv6. By compiling all necessary information and instructions in one place, it ensures a smooth start without the need to search multiple sources or troubleshoot configuration issues.

1.2 What is xv6?

xv6 is a minimalist operating system created by professors at MIT (Massachusetts Institute of Technology) as educational material for an operating systems course. It is a modern reimplementation of the classic UNIX Sixth Edition (V6), originally released by Bell Labs in 1975. xv6 was designed with simplicity and code clarity in mind, making it an ideal tool for learning the fundamentals of operating systems.

1.3 Why learn xv6?

xv6 is an educational tool that has gained recognition for its minimalist approach. Its source code is compact (approximately 10,000 lines of code) and contains only the most basic components of an operating system, allowing for thorough analysis and understanding.

Another advantage of xv6 is its compatibility with the classic UNIX system on which it is based. Many mechanisms and structures implemented in xv6 have equivalents in modern operating systems. Additionally, the code of xv6 is designed to be clear and well-documented, as described in the xv6 book.

xv6 is also a versatile tool that can be run both on real hardware and in an emulator, such as QEMU. This flexibility makes xv6 an excellent choice for experimentation and learning, allowing easy testing and modification of the operating system in a safe environment. xv6 is also considered a great starting point for those planning to delve into the Linux kernel code.

1.4 Where is xv6 used?

xv6 is primarily used in academic environments. It forms the basis of many operating systems courses conducted at universities worldwide, including MIT, Stanford University, and other prestigious institutions.

1.5 RISC-V Architecture

RISC-V is a modern, open, and free processor architecture based on RISC (Reduced Instruction Set Computer) principles. It is modular, meaning its basic instruction set can be

extended with additional functionalities as needed. It finds applications in various fields, such as microcontrollers and supercomputers. Thanks to its growing community and support from many companies, RISC-V is becoming a significant competitor to closed architectures like ARM and x86. The history and applications of the RISC-V architecture are well-presented in the video: Explaining RISC-V: An x86 & ARM Alternative.

2 Environment configuration

To run the `xv6-riscv` system, designed for the RISC-V architecture, we need appropriate tools that enable both translation of the source code into instructions for this architecture and its emulation. The computers we use daily are primarily based on architectures like x86-64 or ARM, making it impossible to run such a system directly without additional tools.

To work with `xv6-riscv`, a dedicated *toolchain* is required—a set of tools that facilitates code compilation for the target architecture and its emulation. This toolchain includes a dedicated compiler for RISC-V (`riscv-gnu-toolchain`), an emulator (e.g., `QEMU`), and additional libraries and support tools. While it is possible to manually install and configure the entire toolchain, this approach can be problematic and time-consuming. Installing each component requires a thorough understanding of their dependencies and the ability to troubleshoot potential installation errors.

An alternative solution is to use a preconfigured system prepared on the `Docker` platform. Docker is a tool that allows applications to run in isolated environments—containers that include all necessary tools and libraries. With Docker, there is no need to manually install additional components on your computer—everything you need is included in a ready-made Docker image. This approach eliminates configuration errors, speeds up the setup process, and allows you to focus immediately on learning the `xv6` system.

In this guide, we will focus on the second solution and, in the following steps, demonstrate how to set up an environment for running `xv6-riscv` using Docker.

2.1 Docker environment configuration

Instructions for installing Docker are widely available on the internet and vary depending on the operating system. Therefore, this section will omit a detailed installation description. These instructions can be found on Docker's official website (`https://docs.docker.com/get-docker/`) or in numerous other online resources.

Assuming Docker is already correctly installed on your computer, we will focus on configuring the Docker image and preparing a container for `xv6-riscv`. This container includes all necessary tools, such as the `riscv-gnu-toolchain` compiler and the `QEMU` emulator, enabling quick startup with the `xv6-riscv` system. The following steps will guide you through downloading and configuring such a container.

Step 1: Download Docker image from Docker Hub

First, download a preconfigured image containing tools for working with the RISC-V architecture. We will use the image `szymongrrr/xv6-riscv-environment:latest`,

available in the Docker Hub repository and based on a Dockerfile provided on GitHub (xv6-riscv-docker).

```
sudo docker pull szymongrrr/xv6-riscv-environment:latest
```

Step 2: Verify the image

To ensure the image was downloaded correctly and works, launch a temporary container:

```
sudo docker run -it szymongrrr/xv6-riscv-environment:latest
```

If the container starts correctly, the terminal should display a prompt indicating you are inside the container environment:

`root@<containerID>:/home/os-iitm#`

After testing, you can terminate the container by pressing Ctrl+D and remove it to keep the system clean. To do this:

1. Display a list of all containers (including stopped ones):

```
sudo docker ps -a
```

2. Find the container ID from the list, then remove it using the command:

```
sudo docker rm <containerID>
```

Replace `<containerID>` with the ID of your container.

2.2 First run of xv6-riscv in the container

To run xv6-riscv, follow these steps:

Clone the xv6-riscv Repository

First, clone the official xv6-riscv repository to the host system:

```
git clone https://github.com/mit-pdos/xv6-riscv
```

Run xv6-riscv in the Docker container

To run xv6-riscv in a container, perform the following steps:

1. Start the Docker container, sharing the cloned `xv6-riscv` directory from the host system with the container:

   ```
   docker run -it -v <path to xv6-riscv folder on host
   ↪  system>:/home/os-iitm/xv6-riscv szymongrrr/xv6-riscv-environment:latest
   ```

   Replace `<path to xv6-riscv folder on host system>` with the path to the xv6-riscv directory obtained using `pwd`. The `-v` flag enables folder sharing between the host system and the Docker container. The `xv6-riscv` repository will thus be accessible in the container.

2. Once inside the container environment (e.g., `root@<containerID>:/home/os-iitm#`), navigate to the `xv6-riscv` directory and start xv6-riscv using the commands:

   ```
   cd xv6-riscv
   make qemu
   ```

If all operations were performed correctly, the terminal should display the following message:

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
```

At this point, the xv6 system is running and ready for use.

To terminate xv6 and the QEMU emulator, press Ctrl+A followed by X. After finishing your work, you can exit the container environment by pressing Ctrl+D, which will log you out and stop the container.

2.3 Restarting the container

To restart a previously created container and continue working in its environment, follow these steps:

1. **Display the list of containers**
   First, check the available containers (including stopped ones):

   ```
   sudo docker ps -a
   ```

   Find the `container ID` of the container you want to restart.

2. **Restart the container in interactive mode**
   To resume work in a stopped container, use the following command:

   ```
   sudo docker start -i <containerID>
   ```

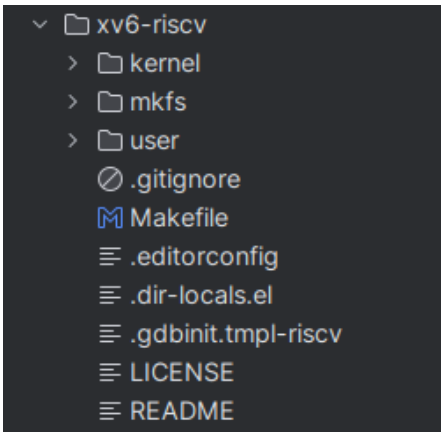
   Replace `<containerID>` with the ID of your container.

3. **Return to the container environment**
   After executing the above command, you will re-enter the container environment (e.g., `root@<containerID>:/home/os-iitm#`), with access to all previously executed operations and changes.

Using these steps, you can resume work in a preconfigured Docker environment without recreating or launching a new container.

3 Introduction to the structure of xv6

The structure of xv6 is based on a division into two spaces: kernel space and user space, each with its unique functions and responsibilities.



Directory structure of the xv6-riscv system

3.1 Kernel space

The kernel space is the core of the xv6 system. This is where all the key mechanisms of the operating system are implemented, enabling the management of hardware resources and processes. The source code for the kernel space is located in the `kernel/` directory. Example files and their functions include:

- `proc.c`: process management, including the implementation of the scheduler.
- `syscall.c`: handling system calls.
- `trap.c`: handling interrupts.

Example code from `proc.c`, showing a simplified implementation of the scheduler:

```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for (;;) {
    intr_on();
    for (p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if (p->state == RUNNABLE) {
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        c->proc = 0;
      }
      release(&p->lock);
    }
  }
}
```

The `scheduler` function iterates through the process table and allocates CPU time to processes in the `RUNNABLE` state.

3.2 User space

The user space is the layer where user programs operate, utilizing functions provided by the kernel. These programs perform various tasks such as browsing directories, editing files, and inter-process communication.

The source code for user programs is located in the `user/` directory. Example programs include:

- `ls.c`: displays the contents of a directory.
- `cat.c`: displays the contents of files.
- `sh.c`: implements the shell.

Example code from the `ls.c` program:

```
int main(int argc, char *argv[]) {
  int fd;
  struct dirent de;
  if ((fd = open(".", O_RDONLY)) < 0) {
    printf("ls: cannot open .\n");
    exit(1);
  }
  while (read(fd, &de, sizeof(de)) == sizeof(de)) {
    if (de.inum == 0)
      continue;
    printf("%s\n", de.name);
  }
```

```
  close(fd);
  exit(0);
}
```

The code reads the contents of the `.` directory and prints the names of the files.

3.3 Shell

The `sh.c` shell is one of the key user programs in `xv6`. It is a simple command interpreter that allows the user to interact with the system. The shell waits for user commands, parses, interprets, and executes them. It can run user programs, manage processes, and pass data between them using pipes.

Key components of the shell include:

- **Main loop**: The shell runs in an infinite loop, displaying a prompt, reading user commands, and processing them.
- **Command parsing**: The `parsecmd()` function splits user input into arguments and identifies operators such as redirections or pipes.
- **Process creation**: The `runcmd()` function uses `fork` to create new processes for executing commands. The child process executes the command using `exec`.

Example code demonstrates the main loop of the shell:

```
int main(void) {
  static char buf[100];
  while (getcmd(buf, sizeof(buf)) >= 0) {
    if (fork() == 0)
      runcmd(parsecmd(buf));
    wait(0);
  }
  exit(0);
}
```

In the code:

- `getcmd`: reads the user command.
- `parsecmd`: parses the input command.
- `runcmd`: executes the command in a child process.

4 System modifications

4.1 Adding a new system call

Adding a new system call to the xv6 kernel requires several steps. Here's how you can add a system call named `kmemfree()`, which returns the amount of free kernel memory. The

instructions for adding a new system call are based on the material: Getting Started with xv6 - User Progs and System Calls.

1. **Define the system call**
   First, you need to define the system call. Open the `kernel/syscall.h` file and add a new system call number at the end of the list. For example:

```
#define SYS_kmemfree 22
```

2. **Implement the system call**
   Next, you need to implement the system call. Create a new function in `kernel/sysproc.c`:

```
uint64
sys_kmemfree(void)
{
  return (uint64) kmemfree();
}
```

   Now, you need to implement this function in `kernel/kalloc.c`:

```
uint64
kmemfree(void)
{
  struct run *r;
  uint64 free = 0;
  for(r = kmem.freelist; r; r=r->next)
    free += PGSIZE;
  return free;
}
```

3. **Add the prototype to `kernel/defs.h`**
   Add the function prototype for `kmemfree()`:

```
// kalloc.c
void*           kalloc(void);
void            kfree(void *);
void            kinit(void);
uint64          kmemfree(void);
```

4. **Add the system call to the system call table**
   Next, you need to add the system call to the system call table. Open `kernel/syscall.c` and add:

```
extern uint64 sys_kmemfree(void);
```

   Then, add a new entry to the system call table:

```
[SYS_kmemfree] sys_kmemfree,
```

5. **Add `kmemfree()` to `user/usys.pl`**

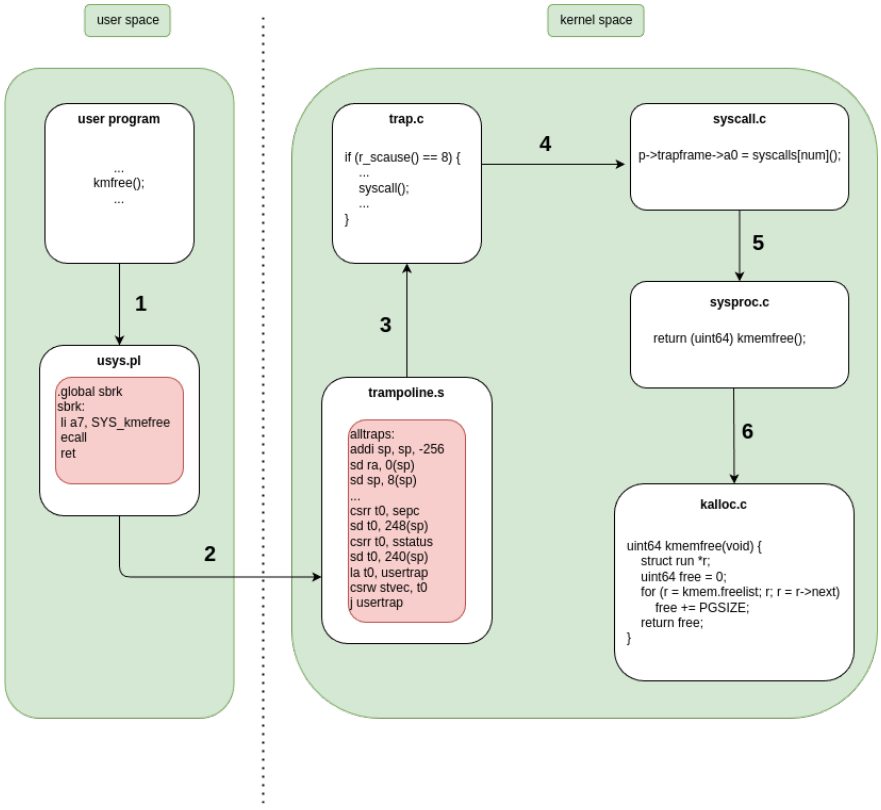
```
entry("kmemfree");
```

6. **Add `kmemfree()` to `user/usys.h`**

```
int kmemfree(void);
```

4.2 Workflow of the system call based on `kmemfree()`

After adding the system call, let's trace its operation. Below is a diagram illustrating how a request from a user program (user space) passes through various layers of the system before reaching the kernel function and then returning with the result.



System call workflow for `kmemfree()`

1. **User program:**

The system call starts in a user program that executes the `kmemfree()` function. This program runs in user space and sends a request to the kernel of the operating system.

```
uint64 free_memory = kmemfree();
```

2. **Wrapper in `usys.pl`:**

The system call is mapped to the corresponding function in the `usys.pl` file. This function sets the system call number in the `a7` register and executes the `ecall` instruction, which generates an interrupt and transfers control to the kernel.

```
.global kmemfree
kmemfree:
    li a7, SYS_kmemfree
    ecall
    ret
```

3. **Trampoline in `trampoline.s`:**

The `trampoline.s` file plays a crucial role in handling transitions between user space and kernel space in xv6. The code in this file is mapped to the same virtual space in both user and kernel modes, allowing efficient context switching for processes.

The `alltraps` function in `trampoline.s` saves the current processor register state onto the process stack and then redirects control to the `usertrap()` function in the kernel. This is the first step in handling a system call or interrupt from user space.

Each process in xv6 has its own instance of this code in kernel space, ensuring security and isolation between processes. Below is a key code fragment from `trampoline.s`:

```
alltraps:
    // Save registers onto the process stack
    addi sp, sp, -256
    sd ra, 0(sp)
    sd sp, 8(sp)
    ...
    csrr t0, sepc  // Get the user program counter
    sd t0, 248(sp)
    csrr t0, sstatus
    sd t0, 240(sp)

    // Redirect control to the kernel handler
    la t0, usertrap
    csrw stvec, t0
    j usertrap
```

4. **Handler in `trap.c`:**

The `usertrap()` function in `kernel/trap.c` handles interrupts and exceptions generated by user processes. In the case of a system call (determined by `r_scause() == 8`), it redirects control to the `syscall()` function.

```
// Save the user program counter
p->trapframe->epc = r_sepc();

if (r_scause() == 8) {  // System call
    if (killed(p))
        exit(-1);
```

```
    // Move the program counter to skip the ecall instruction
    p->trapframe->epc += 4;

    // Enable interrupts after handling registers
    intr_on();

    // Handle the system call
    syscall();
}
   }
```

5. **Dispatcher in `syscall.c`:**

The `syscall.c` file acts as the dispatcher for system calls. It contains the `syscalls[]` table, which maps system call numbers to their corresponding functions. The implementations of these functions are located in files relevant to their area of responsibility, such as `sysproc.c` (processes) or `sysfile.c` (file operations). The `syscall()` function reads the system call number from the register and invokes the appropriate function from the global `syscalls[]` table.

```
   extern uint64 sys_kmemfree(void);
   static uint64 (*syscalls[])(void) = {
       ...
       [SYS_kmemfree] sys_kmemfree,
   };
```

```
void
syscall(void)
{
  int num; // System call number
  struct proc *p = myproc(); // Get the currently running process

  // Read the system call number from the a7 register
  num = p->trapframe->a7;

  // Check if the system call number is valid (within the table range)
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Call the corresponding system call function from syscalls[]
    // Store the function's result in register a0 for access by the user program
    p->trapframe->a0 = syscalls[num]();
  } else {
    // If the system call number is invalid, log a warning and return -1
    printf("%d %s: unknown sys call %d\n", p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

6. **Function in `sysproc.c`:**

The `sys_kmemfree()` function implements the logic of the system call by invoking the `kmemfree()` function, which is implemented in `kalloc.c` and calculates the amount of available memory.

```
uint64 sys_kmemfree(void) {
    return (uint64) kmemfree();
}
```

7. **Actual implementation of the `kmemfree()` function in `kalloc.c`:**
The `kmemfree()` function traverses the list of free memory pages (`freelist`) and calculates their total size.

```
uint64 kmemfree(void) {
    struct run *r;
    uint64 free = 0;
    for (r = kmem.freelist; r; r = r->next)
        free += PGSIZE;
    return free;
}
```

8. **Return to the user program:**

After completing the system call, the operating system must return the result of the operation to the user program and allow it to continue execution.

The first step is to save the result of the system call in the `a0` register. This register serves as the standard location for storing the results of system calls in the RISC-V architecture. For the `kmemfree()` system call, the `a0` register will contain the number of bytes available in the free kernel memory.

Next, the system must restore the full context of the user process that invoked the `kmemfree()` system call so that it can continue execution from the exact point where it was interrupted. The context includes, among other things, the state of the registers, the program counter (PC), the stack pointer, and other process-related data. A key step is setting the program counter (`PC`) to the address of the next instruction to be executed, which in xv6 means the instruction immediately after the `ecall`.

The following step is switching the system from kernel mode to user mode. At this stage, the user process is isolated from direct access to system resources, ensuring the safety of both the program and the operating system. User mode allows the program to continue execution while restricting its ability to interfere with kernel operations.

This process concludes with resuming the execution of the user program. The program counter (`PC`) points to the instruction to be executed after the system call, and the program can read the operation result from the `a0` register. For example, if the user invoked the `kmemfree()` function, the result (number of bytes of free memory) will be directly available in this register.

In xv6, this process is handled by the `usertrapret()` function, which restores the user program context, switches from kernel mode to user mode, and sets the appropriate register values. Below is the code fragment responsible for advancing the program counter (`PC`) and resuming the process in the `/kernel/trap.c` file within the `usertrap()` function.

```
p->trapframe->epc += 4; // Skip the ecall instruction
usertrapret(); // Restore user context
```

Through these mechanisms, xv6 ensures safe and seamless transitions between user space and kernel space, fulfilling the core principles of modern operating systems based on the UNIX architecture.

4.3 Adding a new user program

We can now create a new user program that invokes the added system call.

1. **Create a new file `user/kmemfree.c`:**
   Write the following code to implement the user program:

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  printf("Free kernel memory: %d\n", kmemfree());
  exit(0);
}
```

2. **Add the user program to the Makefile:**
   Open the `Makefile` and add `kmemfree` to the `UPROGS` list:

```
UPROGS=\
    _cat\
    ...
    _kmemfree\
    ...
```

5 Debugging xv6

Debugging xv6 requires the use of the GDB debugger, which connects to the QEMU emulator through a remote connection on localhost. GDB provides the ability to analyze processor registers, memory, and program flow, which is essential for understanding system behavior. This allows you to pause program execution, step through the code, and troubleshoot potential issues.

QEMU offers a debugging server that listens on a local port, e.g., `localhost:25000`. A GDB client, launched in another shell within the same container, connects to this server, enabling access to the state of the simulated xv6 environment and sending debugging commands. This configuration maintains isolation between the emulator and the debugger, simplifying the debugging process.

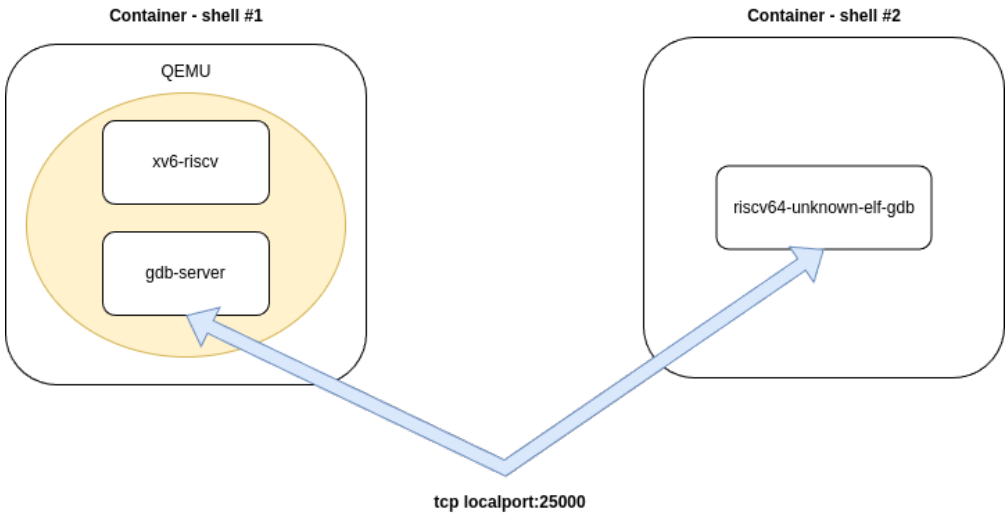


Diagram of debugging xv6 using a remote GDB connection

The following steps will demonstrate how to launch the QEMU emulator with xv6 and a GDB server listening on a localhost port.

5.1 Configuring GDB

1. Launching QEMU with the GDB server

To facilitate debugging, you should add the `-O0` flag, which disables compiler optimizations, allowing for more accurate mapping of the source code in the debugger.

In the `Makefile`, you can use a condition to automatically add the `-O0` flag only when the compilation target is `qemu-gdb`. Example configuration:

```
# Disable compiler optimizations for qemu-gdb mode
ifeq ($(MAKECMDGOALS),qemu-gdb)
CFLAGS += -O0
endif
```

Next, within the container, launch the QEMU emulator in debug mode:

```
make qemu-gdb
```

The emulator will pause xv6 execution at the first instruction, waiting for a GDB client connection.

A message similar to the following will appear in the terminal:

```
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3
↪  -nographic -global virtio-mmio.force-legacy=false -drive
↪  file=fs.img,if=none,format=raw,id=x0 -device
↪  virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb tcp::25000
```

The line `-gdb tcp::25000` indicates that the GDB server is listening on TCP port 25000, allowing a GDB client to connect to the QEMU instance via the GDB server.

2. Launching the GDB client in a new shell

On the host system, open a new shell and check the running containers:

```
sudo docker ps
```

Find the container ID from the list, then start another shell within that container and launch the GDB client:

```
sudo docker exec -it <container_id> bash
riscv64-unknown-elf-gdb
```

3. Connecting GDB to QEMU

In the GDB client, connect to the debugging server on the port specified in step 1. In our case, this is TCP port 25000.

```
target remote localhost:25000
```

4. Debugging

After connecting to the GDB server, you can begin debugging the system code. Depending on whether you want to debug the kernel code (`kernel/`) or user programs (`user/`), it is necessary to load the appropriate binary file.

The kernel code (`kernel/`) is contained in a single binary file, `kernel/kernel`, which includes all the compiled code in the kernel space. User programs (`user/`) have separate binary files corresponding to individual programs.

Debugging kernel code (`kernel space`)

To debug files in the `kernel/` directory, follow these steps:

1. Load the kernel binary file in GDB:

```
file xv6-riscv/kernel/kernel
```

2. Set a breakpoint in a selected function, e.g., `syscall()` in the file `kernel/syscall.c`:

```
b syscall
```

3. Resume program execution:

```
c
```

GDB will halt the program execution every time it enters the `syscall` function.

Debugging user code (`user space`)

If you want to debug files in the `user/` directory, follow these steps:

1. Load the corresponding user program binary file in GDB. For example, to debug the `ls` program in the file `user/ls.c`:

```
file xv6-riscv/user/_ls
```

The file `user/_ls` corresponds to the `ls` program.

2. Set a breakpoint in the desired function, e.g., `ls`:

```
b ls
```

3. Resume program execution:

```
c
```

This way, GDB will halt the program execution every time it enters the `ls` function, allowing you to analyze the behavior of the user program.

In summary, when debugging xv6, always select the appropriate binary file:

- If debugging kernel code, load `kernel/kernel`.
- If debugging a user program, load the binary file corresponding to that program (e.g., `user/_ls` for the `ls` program).

Choosing the correct binary file and function for debugging is crucial for GDB to correctly interpret the code and halt program execution at the appropriate points.

6. Ending debugging

To end the debugging session:

- (shell #2) Exit GDB by typing `quit`.
- (shell #1) Stop QEMU using the key combination `Ctrl+A`, followed by `x`.

5.2 Most useful GDB commands

1. **Starting the program and controlling execution:**
   - `run (r)` - Starts the program from the beginning.
   - `continue (c)` - Resumes program execution until a breakpoint is encountered.
   - `step (s)` - Steps into the next instruction, including function calls.
   - `next (n)` - Moves to the next instruction, skipping function details.
   - `finish` - Completes the current function and returns to the caller.

- `quit (q)` - Exits GDB.

2. **Working with breakpoints:**
   - `break (b) <location>` - Sets a breakpoint at a specified location, e.g., a function or line of code (`b main`, `b kernel.c:42`).
   - `info breakpoints` - Displays a list of all set breakpoints.
   - `delete <breakpoint-number>` - Removes a breakpoint by its number.
   - `disable <breakpoint-number>` - Temporarily disables a breakpoint.
   - `enable <breakpoint-number>` - Re-enables a disabled breakpoint.
3. **Inspecting variables:**
   - `print (p) <variable>` - Displays the value of a variable (`p x`).
   - `print <expression>` - Displays the result of an expression (`p x + y`).
   - `info locals` - Shows all local variables in the current function scope.
   - `set <variable> = <value>` - Changes the value of a variable during debugging (`set x = 5`).
4. **Working with memory:**
   - `x/<format> <address>` - Displays memory content at a specified address. The format determines the display type, e.g., `x/4xw 0x1234` (4 words in hexadecimal).
   - `info registers` - Displays the values of all processor registers.
   - `info address <symbol>` - Shows the memory address of a specified symbol (e.g., variable or function).
5. **Source code navigation:**
   - `list (l)` - Displays a portion of the source code at the current program stop.
   - `list <function>` - Shows the source code of a specific function.
   - `info line <line-number>` - Provides information about a line of code, such as its memory address.
6. **Additional commands:**
   - `help <command>` - Displays detailed information about a command.
   - `info threads` - Shows a list of active threads in the program.
   - `thread <thread-number>` - Switches the debugging context to a selected thread.

6 Educational materials from MIT and other resources

The best way to learn the design and operation of xv6 is by using the educational materials provided by the Massachusetts Institute of Technology (MIT). These resources, created by the authors of xv6, are tailored to meet the needs of students, making them an excellent starting point. In addition to MIT resources, there are high-quality online materials prepared by professors and operating system enthusiasts that complement the knowledge and help to better understand xv6 and the RISC-V architecture.

1. **MIT resources**

   - **Official xv6 repository:**
     github.com/mit-pdos/xv6-riscv
   - **xv6 documentation:**
     xv6/book-riscv.pdf
     A detailed book describing the xv6 system, divided into chapters. While it requires some dedication, it provides valuable insights that can be applied to learning other operating systems.
   - **MIT labs:**
     pdos.csail.mit.edu/6.828/2024/
     The *Labs* section contains assignments on various aspects of xv6. Each task is marked with a difficulty level and includes references to relevant documentation chapters. These labs provide practical understanding of the system and require in-depth engagement with the material.

2. **Other resources**

   - **Theoretical explanations of xv6 mechanisms:**
     A playlist of videos focusing on explaining how specific mechanisms in xv6 work:
     xv6 Kernel-1: Intro and Overview
   - **Source code analysis of xv6:**
     A playlist of videos discussing xv6 at the source code level:
     How does an OS boot? // Source Dive // 001

In our opinion, the best way to learn xv6 is by combining several approaches: performing the labs, reading the *xv6* book, and watching educational videos. This method ensures that, in addition to theoretical knowledge, you also gain practical skills. The labs allow you to apply theory in practice, helping to better understand the system's operation. The *xv6* book provides a solid theoretical foundation, explaining key concepts and implementation details of xv6 along with mechanisms used in modern operating systems. Educational videos simplify more complex topics and present them in an accessible way. Such a combination enables a deeper understanding of xv6 and the broader topics of operating systems and RISC-V architecture.