Shahriar Attar - 810100186 Sobhan Alaeddini - 810100188 Matin Bazrafshan - 810100093

Repository Link: https://github.com/Shahriar-0/Operating-System-Lab-Projects-F2024
Latest Commit Hash: `a93044dd11e070a1c0da6a0f7adf03c4d0bd3ae5`

Introduction to xv6 operating system

1. xv6 architecture

xv6 is a modern reimplementation of Sixth Edition Unix (Unix v6) in ANSI C for multiprocessor x86 and RISC-V[1] systems. The purpose of xv6 was to replace the original V6 source code with a modern replacement, as PDP-11 machines are not widely available and the original operating system was written in archaic pre-ANSI C1. In the `x86.h` file you can see commands for x86 processors, also in `asm.h` and `mmu.h`, which both are basic headers, you can see some commands that are specifically used for x86 processors.

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.

2. Process overview in xv6

A process, also referred to as a task, is an executing (i.e., running) instance of a program. Each process has its own isolated address space and resources, and it can execute independently while also interacting with other processes through system calls. Every process in xv6 consists of two main parts:

1. user-space memory which contains three parts, 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.
2. A per-process state which is private to the kernel. Processes in xv6 are managed by the kernel, which is responsible for process scheduling, inter-process communication, memory management, and other critical tasks. To enhance understanding, code implementation is provided (can be found in `proc.c:37`) This code will be explained in detail in other parts:

[1] RISC-V support was added in 2022.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this
process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Code 2.1

3. 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. As you can see in the code, the kernel associates a process identifier, called pid, with any process, saving it in the struct of a process which is accessible any time it needed.

3. File descriptor concept and Pipe in xv6

A file descriptor is an integer number representing a kernel-managed object which a process can either read from or write to it. An important note should be considered is that a file descriptor refers not only to a “file” but also any objects. The interface of a file descriptor abstracts away the differences between files, pipes, and devices, making them all look like streams of bytes.

Every process has its own open files, an integer is assigned to any opened file. All file descriptors and pointers to files will be saved in the “file table” which is in per-process structure. By convention a process reads from file descriptor 0 which representing stdin[2], writes to file descriptor 1 which representing stdout[3] and writes error messages to file

[2] standard input
[3] standard output

descriptor 2 which representing stderr[4], but a process can choose any opened file to read from or write to by referencing its file descriptor.

Pipes provide a way for processes to communicate, in fact pipe is a small kernel buffer which connects output of a process to input of another process. Writing data to one end of the pipe makes that data available for reading from the other end of the pipe. Pipe creates two connected file descriptors, for left process it closes read end of pipe and sets write end of pipe as stdout. For the right process it closes the write end of the pipe and sets the read end of it as stdin. There can be multiple pipes, so a process tree will be created. It is considerable that right process will wait for input until we reach EOF[5] or write-end of the pipe to be closed.

4. `fork` and `exec` system calls

The fork system call is used for creating a new process, the caller process is named parent process, and the called process is named child process. The child process has exactly the same memory contents as the parent process including instruction, data and stack. Although the child has the same memory contents as the parent initially, the parent and child are executing with different memory and different registers: changing a variable in one does not affect the other.

After a process calls fork, a new process will be created and starts at the same line but with a different workspace. after we call `fork()`, it returns pid. The pid could have three possible amounts:

1. pid > 0: It means we are in the parent process, so the `fork()` has returned the pid of the child process.
2. pid = 0: It means we are in the child process, the child is just created so it has a unique pid which is returned to the parent process but according to the fact that the child process starts at the same line, the kernel decides to return 0 to illustrate we are in the child process.
3. pid < 0: it means `fork()` failed, and a new process was not created.

Looking to implemented code for main loop in `init.c:20` can illustrate functionality of fork:

[4] standard error
[5] End Of File

```
for (;;) {
        printf(1, "init: starting sh\n");
        printf(1, "1. Matin Bazrafshan\n2. Shahriar Attar\n3. Sobhan
Alaeddini\n");
        pid = fork();
        if (pid < 0) {
            printf(1, "init: fork failed\n");
            exit();
        }
        if (pid == 0) {
            exec("sh", argv);
            printf(1, "init: exec sh failed\n");
            exit();
        }
        while ((wpid = wait()) >= 0 && wpid != pid)
            printf(1, "zombie!\n");
    }
```

Code 4.1

The exec system call is responsible for replacing the current process's memory image with 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, and so on. xv6 uses the ELF format (ELF format will be discussed later in this section). So to summarize, when a process calls exec, it loads a new program into its address space and starts executing it from the beginning. Although fork copies the parent's file descriptor table along with its memory, it preserves the file table.

The advantage of separating fork and exec is that we can use fork to create a child process then use open, close, dup in the child to change the standard input and output file descriptors, and then execute it using exec. If fork and exec were combined into a single system call, a more complex scheme would be required for the shell to redirect standard input and output, or the program itself would have to understand how to redirect I/O. So some kind of abstraction happens for exec according to the fact that exec does not concern about input and output initially.

Adding a greeting Boot Message

As you can see in Code 4.1, just we need to add a single line:

```
SeaBIOS (version 1.16.2-debian-1.16.2-1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCF250+1EF0F250 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
Group #17:
- Sobhan Alaedini
- Shahriar Attar
- Matin Bazrafshan
$
```

Additional features for xv6 console

- **Ctrl + B: Moving cursor to left, Ctrl + F: Moving cursor to right**

  To perform this action, first we add a `shift` variable to input structure, which shows times that cursor has been shifted to left, it is obvious that shift can't be less than 0, because we can not move cursor further(to the right) than last entered character, and also it can be less than (`input.e - input.w`), which represents first and last part of current command in order. After that we can easily increase or decrease `input.shift` value, then re-set the cursor position according to value shift.

  The tricky part is when we want to put or delete a character, thanks to the `input.shift`, first we shift buffer characters one to right, but not all of them, only ones that range from current position to the end. Then we put a character in the right place and after that we update the console according to `input.buf` and finally set the cursor to its right position. Similarly for deleting a character we can shift buffer to left(only elements ranged within end and current cursor position) and repeat the same process.

```
static void
movpostoleft(void) {
    setpos(getpos() - 1);
}

static void
movpostoright(void) {
    setpos(getpos() + 1);
}
```

```
case C('B'):
    if (input.shift < input.e - input.w) {
        input.shift++;
        movpostoleft();
    }
    break;

case C('F'):
    if (input.shift > 0) {
        input.shift--;
        movpostoright();
    }
    break;
```

- **Ctrl + L: Clearing terminal page and getting ready for new command**

  To perform this, first we need to kill the line which also clears the input buffer, then delete all characters and finally write a '$' in the first position of the console, and set the cursor position to 2.

```
// erase line and clear input buffer
static void
conseraseline(void) {
    movpostoend();
    input.shift = 0;

    while (input.e != input.w && input.buf[(input.e - 1) % INPUT_BUF] !=
'\n') {
        input.e--;
        consputc(BACKSPACE);
    }
}

// erase terminal screen
static void
consclear(void) {
    int pos;
    pos = getpos();
    while (pos >= 0)
        conserasechar(pos--);
}

// print shell prompt
static void
consnewcommand(void) {
    conswritechar(0, '$');
    setpos(2);
}
```

```
case C('L'):
    conseraseline();
    consclear();
    consnewcommand();
    break;
```

- **↑: Restore the last entered commands, ↓: To undo arrow up**

  Before implementing these commands, we need to define a data structure to hold commands and other information about the history of commands. A queue will suit well in this case (the usage will be like stack but since we have arrow down functionality too we need a data structure like queue). After pressing the enter button, the current command will be added to the first of the queue for later needs, increasing the number of entered commands (which is limited to commands buffer size, in our case 10), and resetting all controlling variables (which job is to track every single change in current console command).

  A helpful feature also implemented in this structure is a temporary variable to cache the non-entered command after arrowing up and then reload it back to terminal when the arrow down navigator hits the ground!

```
// buffer (history) of entered commands
#define COMMAND_BUF 10
struct {
    char buf[COMMAND_BUF][INPUT_BUF]; // buffer
    int r;                            // range[1,10], read index
    int w;                            // write index
    int intab;                        // whether we are in tab mode
    char tmpcmd[INPUT_BUF];           // temporary command
    int lastusedidx;                  // index of last used command
} cmds;
```

```c
static void
storecmd(void) {
    for (int i = cmds.w - 1; i > 0; i--)
        for (int j = 0; j < INPUT_BUF; j++)
            cmds.buf[i][j] = cmds.buf[i - 1][j];
    int j = 0;
    for (int i = input.w; i < input.e; i++) {
        cmds.buf[0][j] = input.buf[i];
        j++;
    }
    for (; j < INPUT_BUF; j++) {
        cmds.buf[0][j] = 0;
    }
}

static void
loadcmd(void) {
    conseraseline();
    int n = cmds.r - 1;
    for (int i = 0; i < INPUT_BUF; i++) {
        if (cmds.buf[n][i] == 0)
            break;
        input.buf[input.e++ % INPUT_BUF] = cmds.buf[n][i];
        consputc(cmds.buf[n][i]);
    }
}
```

```c
case ARROW_UP:
    if (cmds.r == 0)
        copycmd();
    cmds.r++;
    if (cmds.r > cmds.w) cmds.r = cmds.w;
    else                 loadcmd();
    break;

case ARROW_DOWN:
    cmds.r--;
    if (cmds.r > 0)
        loadcmd();
    else {
        cmds.r = 0;
        recovercmd();
    }
    break;
```

Some additional features:

- Tab: suggest a word by tracking entered commands
- Ctrl + A: move cursor to the beginning
- Ctrl + E: move cursor to the end
- Ctrl + N: removes all numeric character of console command

All implementations of these functionalities can be found in `console.c` and `consoleintr` function.

Running a user program

The `strdiff.c` file implements a specific-design string comparator, which compares characters one by one and if a character comes first in order, it puts a 1 either it puts a 0 in the destined file.

The importance of coding user programs for xv6 OS is to use xv6 kernel's system calls properly. You can see code below to figure out how system calls are used properly in the main function of strdiff program.

```
int main(int argc, char* argv[]) {
    if (argc != 3) {
        printf(2, "Please enter exactly two strings!\n");
        exit();
    }
    if ((strlen(argv[1]) > 15) || (strlen(argv[2]) > 15)) {
        printf(2, "Length of strings must be equal or less than 15!\n");
        exit();
    }

    unlink("strdiff_result.txt"); // remove links of any file to
strdiff_result.txt

    int fd = open("strdiff_result.txt", O_CREATE | O_WRONLY); // create
or open file
    if (fd < 0) {
        printf(2, "Error happens when trying making file!\n");
        exit();
    }

    if (strdiff(argv[1], argv[2]) == 0) {
        printf(2, "String must only include alphabetical
characters!\n");
        exit();
    }

    write(fd, diff, strlen(diff));
    close(fd);

    exit();
}
```

```
$ strdiff apple banana
$ cat strdiff_result.txt
100011
$ strdiff aaaaaaaaaaaaaaaa b
Length of strings must be equal or less than 15!
$ strdiff a
Please enter exactly two strings!
$ strdiff 11 aa
String must only include alphabetical characters!
$
```

Introduction to operating system and xv6

5. Operating System main responsibilities

An operating system has three major responsibilities:

1. Abstract low-level hardwares for applications, so an application need not be concerned about handling memory, I/O, etc.
2. Using time-share protocols, it shares hardware between applications according to criticalness, priority, etc. It also must ensure that an application does not overuse resources or waste them.
3. It provides controlled ways for programs to interact, so that they can share data or work together.

6. xv6 files

Main files of xv6 will be shortly explained:

- **Basic Headers**: These files define fundamental data types, constants, and function prototypes that are used throughout the xv6 codebase.
- **Locks**: These files implement synchronization mechanisms such as spinlocks and sleeplocks. These locks are used to ensure that multiple threads or processes can safely access shared resources without interfering with each other.
- **Processes**: These files handle process management in xv6. They implement functions for creating, scheduling, and switching between processes. They also contain the necessary code for loading and executing programs in processes.

- **System Calls**: These files implement the system call handler and provide the kernel functions associated with each system call. System calls allow user programs to request privileged operations from the kernel, such as file operations or process management.
- **File System**: These files implement the file system layer in xv6. They provide functions for managing files, directories, and disk I/O operations. They handle operations such as reading from and writing to files, creating and deleting files, and navigating directories.
- **Pipes**: These files implement the pipe mechanism, which allows inter-process communication. Pipes provide a way for processes to communicate by sharing a common buffer, allowing one process to write data that another process can read.
- **String Operations**: These files provide utility functions for string manipulation. They implement common operations on strings, such as copying, concatenating, and comparing strings.
- **Low-Level Hardware**: These files interact with the hardware at a low level. They handle block-level disk I/O operations, provide the driver for the disk interface, and manage console input and output.
- **User-Level**: These files contain the user-level interface of xv6. They include assembly code for user-level system calls, utility functions for user programs, and a library of functions that can be used by user programs.
- **Bootloader**: These files are responsible for bootstrapping xv6. They contain the code for the bootloader, which loads the xv6 kernel into memory and starts its execution.
- **Link**: This file specifies the memory layout of the kernel and how the object files should be linked together during the compilation process.

Name of folders in Linux:

- kernel : /kernel
- header files: /include
- file systems: /fs

xv6 compilation

7. Kernel make

after commanding make -n, we can see:

```
(base) fabulousmatin@MyUbuntu:~/Operating-System-Lab-Projects-F2024/xv6-public$ make -n
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno
-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -o console.o console.c
ld -m    elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o io
apic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o proc.o sleeplock.o spinlock.
o string.o swtch.o syscall.o sysfile.o sysproc.o trapasm.o trap.o uart.o vectors.o vm.o  -b
binary initcode entryother
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
dd if=bootblock of=xv6.img conv=notrunc
dd if=kernel of=xv6.img seek=1 conv=notrunc
```

In this command, the `-o` kernel option specifies the output file name, which is typically kernel. This indicates that the final kernel file is generated by linking the object files using the `ld` command.

8. UPROGS and ULIB in Makefile

`UPROGS`, which stands for user programs, is a list of user programs which will be built and included in the xv6 operating system. During the build process, the Makefile will compile each user program using the `gcc` command and link them with the user-level library functions and system call wrappers provided by xv6. The resulting binary files for these programs will be included in the final xv6 image, allowing them to be executed by user processes within the operating system.

`ULIB` refers to user-level library and is a set of utility functions provided to user programs running within the xv6 operating system. The `ULIB` functions are defined in the `ulib.c` file. `ulib.c` contains some useful C functions and by including it, user programs can perform various operations. They can be described as prepared functions for user programs to interact with the kernel. There is no need to manually add and compile these functions due to xv6 linking them to make rules using ld command.

QEMU

9. QEMU input disks and their data

- xv6.img: This is the disk image file that contains the xv6 operating system. It includes the kernel and all the system-level code necessary for xv6 to run.
- fs.img: This is an additional disk image file that represents a file system. It's used to store user-level programs and data.

xv6 Booting

Bootloader

10. Data in first sector of bootable disk

The first commands executed by the Makefile include compiling the object files `bootmain.c` and `bootasm.S`, linking these two to produce `bootblock.o`, and using `objcopy` to copy the `.text` section of the `bootblock.o` file to the `bootblock` file. Finally, it's passed to the `sign.pl` script to add a 2-byte boot signature to `bootblock`.

In the first sector (the first 512 bytes) of the bootable disk, the contents of the `bootblock` file are located.

11. Comparing boot binary file with other binary files of xv6 and converting it to assembly

In xv6, the binary objects are of the type **ELF** [6], defined in `elf.h`. This format is used for object files, file libraries, and executables. It's a standard binary format for Unix and Unix-like systems. The ELF format has replaced older executable formats in various environments. It provides robust, flexible, and efficient facilities for linking and loading.

It was formerly named Extensible Linking Format and is a common standard file format for executable files, object code, shared libraries, and core dumps. It was first published in the specification for the application binary interface (ABI) of the Unix operating system version named System V Release 4 (SVR4), and later in the Tool Interface Standard.

ELF files are typically the output of a compiler or linker and are a binary format. They are designed to be flexible, extensible, and cross-platform. For instance, they support different endiannesses and address sizes so they do not exclude any particular central processing unit (CPU) or instruction set architecture. This has allowed it to be adopted by many different operating systems on many different hardware platforms.

[6] Executable and Linkable Format

Each ELF file is made up of one ELF header, followed by file data. The data can include:

- **program header table**: describing zero or more memory segments.

- **section header table**: describing zero or more sections.

- **Data**: referred to by entries in the program header table or section header table.

In xv6 it consists of these parts:

- **ELF header:** (`struct elfhdr`, in this header there is a section called `e-entry` which defines the entry point address for program)

- **A sequence of program section headers:** (`struct proghdr`). each of proghdr describes a section of the application that must be loaded into memory, namely:

- **.text**: executables commands of the program
- **.rodata**: read-only data, suck as string literals in C
- **.data**: initialized values such as global variables
- **.bss**: not initialized data, because we only have their size and address

The ELF file has two views: the program header shows the segments used at run time, whereas the section header lists the set of sections. This structure allows the operating system to interpret its underlying machine instructions correctly.

An ELF binary starts with the four-byte “magic number” 0x7F, E’, ’L’, ’F’, or `ELF_MAGIC`, so you can easily check whether files are ELF binary or not. Additionally you can use `objdump -h filename.o` command to see the binary file format this is the result for the files in xv6:

```
bio.o:     file format elf32-i386             ||   picirq.o:     file format elf32-i386
bootasm.o:     file format elf32-i386         ||   pipe.o:     file format elf32-i386
bootblock.o:     file format elf32-i386       ||   printf.o:     file format elf32-i386
bootblockother.o:     file format elf32-i386  ||   proc.o:     file format elf32-i386
bootmain.o:     file format elf32-i386        ||   sh.o:     file format elf32-i386
cat.o:     file format elf32-i386             ||   string.o:     file format elf32-i386
console.o:     file format elf32-i386         ||   sysproc.o:     file format elf32-i386
echo.o:     file format elf32-i386            ||   swtch.o:     file format elf32-i386
entry.o:     file format elf32-i386           ||   spinlock.o:     file format elf32-i386
entryother.o:     file format elf32-i386      ||   rm.o:     file format elf32-i386
exec.o:     file format elf32-i386            ||   sleeplock.o:     file format elf32-i386
file.o:     file format elf32-i386            ||   strdiff.o:     file format elf32-i386
forktest.o:     file format elf32-i386        ||   stressfs.o:     file format elf32-i386
fs.o:     file format elf32-i386              ||   syscall.o:     file format elf32-i386
grep.o:     file format elf32-i386            ||   sysfile.o:     file format elf32-i386
ide.o:     file format elf32-i386             ||   trapasm.o:     file format elf32-i386
initcode.o:     file format elf32-i386        ||   trap.o:     file format elf32-i386
init.o:     file format elf32-i386            ||   uart.o:     file format elf32-i386
ioapic.o:     file format elf32-i386          ||   ulib.o:     file format elf32-i386
kalloc.o:     file format elf32-i386          ||   umalloc.o:     file format elf32-i386
kbd.o:     file format elf32-i386             ||   usertests.o:     file format elf32-i386
kill.o:     file format elf32-i386            ||   usys.o:     file format elf32-i386
lapic.o:     file format elf32-i386           ||   vectors.o:     file format elf32-i386
ln.o:     file format elf32-i386              ||   vm.o:     file format elf32-i386
log.o:     file format elf32-i386             ||   wc.o:     file format elf32-i386
ls.o:     file format elf32-i386              ||   zombie.o:     file format elf32-i386
main.o:     file format elf32-i386            ||
mkdir.o:     file format elf32-i386           ||
mp.o:     file format elf32-i386              ||
```

All of them, including `bootblock.o` (bootloader), are elf32-i386. The complete output is more detailed but we just show the result for `bookblock.o` and `bio.o` for this discussion.

```
bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001c3  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007dc4  00007dc4  00000238  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      0000001e  00000000  00000000  000002e8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000308  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   0000058f  00000000  00000000  00000348  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 00000232  00000000  00000000  000008d7  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   00000281  00000000  00000000  00000b09  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    00000210  00000000  00000000  00000d8a  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line_str 00000064  00000000  00000000  00000f9a  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 00000198  00000000  00000000  00000ffe  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000033  00000000  00000000  00001196  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

```
bio.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000023b  00000000  00000000  00000040  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000027b  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00004958  00000000  00000000  00000280  2**5
                  ALLOC
  3 .rodata.str1.1 0000002d  00000000  00000000  00000280  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .debug_info   00000459  00000000  00000000  000002ad  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000025e  00000000  00000000  00000706  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_loclists 000000e5  00000000  00000000  00000964  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 00000020  00000000  00000000  00000a49  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  8 .debug_rnglists 00000017  00000000  00000000  00000a69  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_line   0000024d  00000000  00000000  00000a80  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
 10 .debug_str    000001c9  00000000  00000000  00000ccd  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .debug_line_str 000000b4  00000000  00000000  00000e96  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 12 .comment      0000001f  00000000  00000000  00000f4a  2**0
                  CONTENTS, READONLY
 13 .note.GNU-stack 00000000  00000000  00000000  00000f69  2**0
                  CONTENTS, READONLY
 14 .eh_frame     000000d0  00000000  00000000  00000f6c  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

But the difference between a boot binary file with others is that it doesn't contain `.data` or `.bss` and the main part is `.text`. The reason is that bootloader have a constant address (namely 0x7C00) so the only thing that matters is the code that has to run. The address specification is done in `Makefile` by the -Ttext flag like this. Also the `-e start` specifies that the label `start` in assembly code (in our case `booasm.S`) is the beginning of the program.

```
bootblock: bootasm.S bootmain.c
	$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
	$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
	$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o
bootasm.o bootmain.o
	$(OBJDUMP) -S bootblock.o > bootblock.asm
	$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
	./sign.pl bootblock
```

So the `bootblock` file is created on one to the last line of the `bootlock` target in makefile and then boot signatures are added to it by `sign.pl` file. The flags and functionality of this line will be explained shortly in the next question, but to summarize it this line puts the `.text` section of `bootblock.c` file in `bootblock` as raw binary so the `bootblock` does not comply with the ELF format, hence it doesn't have header or other redundant data. So the boot file format is raw binary which contains only instructions for x86 architecture. There are few reason to why it doesn't follow the ELF format:

1. **Viability**: CPU doesn't know the ELF format. When we want to boot the system we still haven't loaded the kernel therefore we don't know ELF. If BIOS passes the boot file as ELF then the CPU would consider everything in the file as instruction (even headers or other non-related data) and may cause damage.
2. **Size Limitation**: The bootblock must fit within a single sector (512 bytes) on the disk1. This is a limitation of the BIOS, which loads the first sector of the boot disk into memory and starts executing it
3. **Simplicity**: The bootblock's job is to load more data of the kernel instructions from disk1. It's left to OS developer to make it work

For converting the bootblock to assembly we used `objdump` command. This is a command-line program used to display various information about object files on Unix-like operating systems, but since it's raw binary and we don't know anything about architecture and sections we have to pass flags manual to `objdump`. The final command was this:

```
objdump -D -b binary -m i386 -M addr16,data16 -adjust-vma=0x7C00
bootblock > bootblock.S
```

- `-D`: This option tells `objdump` to disassemble all sections that have machine code, not just the .text section1.
- `-b binary`: This option specifies the input format for objdump. In this case, it's set to binary, which means the input file is a raw binary file without any specific format like ELF or PE1.

- `-m i386`: This option specifies the architecture of the binaryi386 refers to the Intel 80386, indicating that the code is 32-bit x86 machine code1.
- `-M addr16,data16`: This option is passed to the disassembler `addr16,data16` specifies that the code uses 16-bit addresses and data, which is typical for real-mode x86 code (like a bootloader).
- `-adjust-vma=0x7C00`: To adjust the starting line for output assembly. We used 0x7C00 to imitate the real xv6 bootloader.
- `bootblock`: This is the input file for `objdump`. It's a raw binary file containing x86 machine code.
- `> bootblock.S`: to store the result in an assembly file.

The output file can be seen in the github repository.

12. `objcopy` in make

objcopy is a utility in Unix-like operating systems that copies and translates object files. It uses the GNU BFD[7] Library to read and write the object files. It can write the destination object file in a format different from that of the source object file and for doing that it uses temp files and then delete them. It's exact behavior is controlled by the command-line, some commonly-used flags for this command are:

- `-O binary`: This option tells `objcopy` to output in raw binary format. When `objcopy` generates a raw binary file, it essentially produces a memory dump of the contents of the input object file. All symbols and relocation information will be discarded.
- `-S` or `--strip-all`: This option tells `objcopy` to strip all symbols from the output. Symbols are generally used for debugging and aren't required for execution, so stripping them can decrease the size of the executable. Two main parts are the symbol table and relocation records which will be deleted in the destination file. The symbol table contains entries for variables and functions which are accessed/called from other object files, and the relocation records holds the addresses of all locations of the assembled code which have to be updated during loading.
- `-j .text`: This option tells `objcopy` to only keep the `.text` section of the object file.

In xv6 it's used for few purposes:

1. It's used to create the bootblock. The command `$ (OBJCOPY) -S -O binary -j .text bootblock.o bootblock` takes the object file `bootblock.o`, strips off all symbols (`-S`) and outputs it in binary format (`-O binary`). The `-j .text` option tells

[7] Binary File Descriptor

`objcopy` to only keep the `.text` section of the object file. The output is then stored in the `bootblock`, which is used as the boot block of the xv6 image. Then it's passed to the `sign.pl` file to check it's not more than 510 bites, then two bytes (0x55 and 0xaa which are boot signatures) are added at the end of the file.

2. `objcopy` is used to create `entryother`, which is a .text section of `bootblockother.o` and is loaded at a specific memory location when xv6 boots up.
3. `objcopy` is also used to create `initcode`, which is the first piece of code that runs in a new environment when a new process is created. The `initcode` file is the raw binary of `initcode.out`.

In the end by linking entry.o and other *.o files (stored in `OBJS` variable) and `initcode` and `entryother`, the kernel would be ready.

13. Reason for combining C and assembly for booting xv6

The initial stages of booting involve direct interactions with hardware components, setting up the environment for the operating system. Assembly language provides low-level access to hardware and is capable of performing tasks such as setting up the stack, initializing registers, and switching the processor from real mode to protected mode. These tasks are hardware-specific and require precise control which high-level languages like C cannot provide.

When BIOS loads the boot sector the x86 processor is in real mode and addressing is always done physically rather than usual virtual addressing and we have 16-bit processor and 1 Mbyte memory. In order to use a 32-bit processor and up to 4 Gbyte memory we should switch to protected mode which can only be done in assembly by making the first bit of Control Register equal to 1.

14. x86 registers

There are 4 main types for registers in x86 architecture:

1. **General-Purpose Registers (GPR)**: These are used for various purposes by the command set. The x86 architecture has 8 GPRs, and 64-bit x86 (x86-64) has additional registers. The 8 GPRs are as follows:
    - Accumulator register (AX): Used in arithmetic operations. For instance it can be used to temporary store result of ALU[8]. The reason for calling it, Accumulator, is that after each calculation, the result is stored in this register and for the next calculation it will be used for input and continues to do this loop.
    - Counter register (CX): Used in shift/rotate instructions and loops.
    - Data register (DX): Used in arithmetic operations and I/O operations.
    - Base register (BX): Used as a pointer to data.
    - Stack Pointer register (SP): Pointer to the top of the stack.
    - Stack Base Pointer register (BP): Used to point to the base of the stack.
    - Source Index register (SI): Used as a pointer to a source in stream operations.
    - Destination Index register (DI): Used as a pointer to a destination in stream operations.
2. **Segment Registers**: In the x86 architecture, there are six segment registers.
    - Data Segment (DS): Points to the segment containing data (like variables).
    - Code Segment (CS): Points to the segment containing the current program code
    - Stack Segment (SS): Points to the segment containing the stack.
    - Extra Segment (ES): An additional segment that's generally used by string operations.
    - FS and GS: These are extra segment registers that don't have specific uses defined by the hardware.

[8] Arithmetic Logic Unit

3. **The flag register**: In the x86 architecture, also known as FLAGS, is a status register that contains the current state of the CPU. The size and meanings of the flag bits are architecture dependent. It usually reflects the result of arithmetic operations as well as information about restrictions placed on the CPU operation at the current time. Here are some of the key flags in the FLAGS register:
    - Carry Flag (CF): Set if an arithmetic operation results in a carry (for addition) or borrow (for subtraction).
    - Parity Flag (PF): Set if the number of set bits in the result is even.
    - Auxiliary Carry Flag (AF): Used in BCD[9] arithmetic.
    - Zero Flag (ZF): Set if an operation results in zero.
    - Sign Flag (SF): Set if the result of an operation is negative.
    - Trap Flag (TF): Used for on-off control for single-step debugging.
    - Interrupt Enable Flag (IF): Controls whether hardware interrupts are allowed.
    - Direction Flag (DF): Used by string operations to auto-increment or auto-decrement the index register.
    - Overflow Flag (OF): Set if an arithmetic operation results in a value that overflows its intended range.

In addition to these, there are several other flags used for system-level operations. The FLAGS register is 16-bits wide. Its successors, the EFLAGS and RFLAGS registers, are 32-bits and 64-bits wide, respectively. The wider registers retain compatibility with their smaller predecessors.

4. **Control registers**: In the x86 architecture are special-purpose registers that change or control the general behavior of the CPU. They are used to control operations such as interrupt control, switching the addressing mode, paging control, and coprocessor control. Some of the key control registers in the x86 architecture:

[9] binary-coded decimal

- CR0: The CR0 register is 32 bits long on the 386 and higher processors. On x64 processors in long mode, it (and the other control registers) is 64 bits long. CR0 has various control flags that modify the basic operation of the processor. Each bit in CR0 has special meaning:
    i. Bit 0 (PE: Protected Mode Enable): If this bit is set, the processor is operating in protected mode.
    ii. Bit 1 (MP: Monitor co-processor): Controls interaction with the floating-point unit.
    iii. Bit 2 (EM: x87 FPU Emulation): If this bit is set, x87 FPU instructions are emulated by the CPU.
    iv. Bit 3 (TS: Task switched): Used for task switch handling.
    v. Bit 4 (ET: Extension type): On the 386, it should be set to 1.
    vi. Bit 5 (NE: Numeric error): Controls how floating-point errors are reported.
    vii. Bit 16 (WP: Write protect): Determines whether read-only pages can be written to when in supervisor mode.
    viii. Bit 18 (AM: Alignment mask): Alignment checking of user-mode data accesses when AM bit is set and AC flag (in EFLAGS register) is set.
    ix. Bit 29 (NW: Not-write through): Globally disables write-through caching when set.
    x. Bit 30 (CD: Cache disable): Disables memory cache when set.
    xi. Bit 31 (PG: Paging): Enables paging when set.
- CR2: The CR2 register is used in page-fault exception processing. It contains the linear address that caused a page fault.
- CR3: The CR3 register is used when virtual memory is enabled, and it holds the physical address of the Page Directory.
- CR4: The CR4 register holds control bits for several new features such as Physical Address Extension (PAE), and Page Size Extensions (PSE).

15. x86 mode while booting and it's problems

During the booting process of an x86-based system, the processor starts in a mode called "real mode". This is a 16-bit mode that provides backward compatibility with older x86 processors. However, real mode has several limitations, such as a maximum of 1MB of addressable memory.

To overcome these limitations, modern operating systems quickly switch the processor to "protected mode", which supports 32-bit addressing and provides access to features like virtual memory. In the case of 64-bit systems, the processor is further switched to "long mode" for 64-bit support.

16. Addressing in this mode

In real mode, which is a 16-bit mode present on all x86 processors, the addressing is done using a segmented memory model. This mode is characterized by a 20-bit segmented memory address space, providing exactly 1 MiB (Mebibyte) of addressable memory.

The memory addressing in real mode uses a segment:offset system. There are six 16-bit segment registers: CS, DS, ES, FS, GS, and SS. Segments and offsets are related to physical addresses by the equation: PhysicalAddress = Segment * 16 + Offset. Thus, an address like 12F3:4B27 corresponds to the physical address 0x17A57.

17. Using 0x100000 for kernel address

The address 0x100000 (or 1MB in decimal) is often used as the starting address for loading the kernel in many operating systems, including xv6. This is a convention that dates back to the early days of personal computers. Reason for that are as follows:

1. **Memory Layout**: In the memory layout of an x86-based system, the lower part of the memory (below 1MB) is typically reserved for BIOS, video memory, and other I/O devices. By loading the kernel above 1MB, it avoids overwriting these areas.
2. **Protected Mode**: The x86 processor starts in real mode (16-bit mode) where it can only address 1MB of memory. To use more memory and enable features like virtual memory and multitasking, the processor needs to switch to protected mode. Loading the kernel above 1MB ensures that it's accessible once the processor has switched to protected mode.
3. **Memory Segmentation**: In real mode, memory is segmented and each segment can be a maximum of 64KB. The address 0x100000 is where the first segment that doesn't overlap with another one starts. So, it's a convenient place to load the kernel.

18. `entry.S` equivalent in linux kernel

linux/arch/x86/entry/entry.S at master · torvalds/linux · GitHub

linux/arch/x86/entry/entry_32.S at master · torvalds/linux · GitHub

linux/arch/x86/entry/entry_64.S at master · torvalds/linux · GitHub

Running xv6 kernel

19. Why page table is physical

The page table is stored in physical memory because it's a data structure used by the virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. So if it was virtual we would have needed itself to get the physical address which is in contrast with its very own nature, it's like stucking in an ill-defined dependency cycle which leads to an infinite loop.

20. Explain `entry.S` functions and map them to linux kernel

- **Multiboot Header**: The multiboot header is used by boot loaders like GNU Grub to load the kernel into memory. The magic and flags are specific values defined by the Multiboot Specification.
- **_start**: This symbol specifies the ELF entry point. Since virtual memory hasn't been set up yet, the entry point is the physical address of `entry`.
- **entry**: This is where xv6 starts executing on the boot processor, with paging off. It turns on the page size extension for 4 Mbyte pages, sets up the page directory, turns on paging, sets up the stack pointer, and then jumps to `main()`.
- **stack**: This is a common block of memory reserved for the kernel stack

All of them except entry exists in linux too.

21. Brief explanation about virtual address of kernel

The kernel's virtual address space is set up using a two-level paging mechanism supported by the Memory Management Unit (MMU). The kernel's virtual addresses start from KERNBASE (0x80000000), which is much higher than the user program's addresses. This design allows the same kernel code to be mapped into the virtual address space of every process.

The first part of the kernel's virtual address space from KERNBASE to KERNBASE+4MB is mapped to the same physical addresses, i.e., from 0 to 4MB2. This is done in entry.S using a simple two-entry page directory called `entrypgdir`. The rest of the kernel's virtual address space is set up later in `main.c` by the kvmalloc() function.

22. Reason for using `SEG_USER` in user's data and code

Each part of the kernel and users command has been stored in memory, and are defined by a descriptor in GDT[10] which has data like the beginning of the code, size and CPL[11].

When an instruction is read, the code is found using the GDT and then its page table is found, after converting virtual addressing to physical the instruction is executed. When reading instructions we notice the privilege level either by CPL or DPL[12].

So in other words, in the xv6 operating system, `SEG_USER` is used to differentiate between user space and kernel space. This is crucial for maintaining system security and stability.

The xv6 operating system uses a segmented memory model, where each segment is associated with a specific privilege level. The `SEG_USER` is used to denote segments that belong to user space, which runs with lower privileges compared to kernel space. This ensures that user programs cannot directly access or modify kernel code or data, preventing potential system crashes or security breaches.

When a user program is running, the CPU is set to user mode, and only memory within the `SEG_USER` segment can be accessed. Any attempt to access memory outside this segment, such as kernel memory, will result in a segmentation fault and the program will be terminated.

This mechanism of using `SEG_USER` for user's data and code helps in maintaining the integrity and reliability of the system by protecting the kernel from unauthorized access or modification by user programs. It's a fundamental aspect of operating system design to separate user and kernel spaces[4].

[10] Global Descriptor Table
[11] Current Privilege Level
[12] Descriptor Privilege Level

Running first user program

23. `struct proc` and equivalent in linux

In linux we have task_struct.

The struct proc in xv6 is a data structure that represents a single process. The attributes of this struct are:

- `uint sz`: This represents the size of the memory taken by the process in bytes. It's used to keep track of the process's memory usage.
- `pde_t* pgdir`: This is a pointer to the page table of the process. A page directory entry (PDE) contains information about a particular page table.
- `char* kstack`: This is a pointer to the kernel stack. The kernel stack is part of the kernel space, not user space, and it's used for executing system calls.
- `enum procstate state`: This enum determines the state of the process. It can be in one of several states: UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE.
- `int pid`: This is the PID[13], which is a unique number among all processes.
- `struct proc* parent`: This is a pointer to the parent process (the process that created the current process via the fork function). The type of this pointer is like the current process itself, struct proc.
- `struct trapframe* tf`: This is a pointer to the trap frame for saving the execution state of the program during a system call.
- `struct context* context`: This is a pointer to struct context that holds register values needed for context switching. With the help of the switch function (defined in assembly), you can switch to another process.
- `void* chan`: If its value is not 0, it means that the process is asleep (waiting for something). Here chan means channel, and there are multiple channels including console input channel.
- `int killed`: If its value is not 0, it means that the process has been killed.
- `struct file* ofile[NOFILE]`: This is an array of pointers to files opened by the process.
- `struct inode* cwd`: This variable specifies the current working directory.
- `char name[16]`: The name of the process for debugging purposes.

[13] Process Identifier

24. Why sleep is problematic in system manager code

The sleep function works by releasing a lock before giving up the CPU. This is done to prevent deadlocks and ensure that other processes can acquire the lock while the current process is sleeping. However, this design can lead to race conditions if another process acquires the lock and changes the state of the system before the sleeping process has completely transitioned to the sleeping state

25. Difference between kernel address set by `kvmalloc()` and `setupkvm()`

- `kvmalloc()`: This function sets up the kernel's virtual memory by establishing a two-level paging mechanism. It calls `setupkvm()` to create a new page directory and then maps a range of physical addresses into the page tables in the specified page directory.
- `setupkvm()`: This function is used to set up the kernel's part of the virtual memory when copying the whole virtual memory (user + kernel) from a page directory. It's called during `copyuvm()`, which is used when creating a copy of a process's page table. In contrast, `allocuvm()` extends existing virtual memory (specifically the heap portion), and since there already exists kernel portion of mappings in `allocuvm()`, it doesn't need to call `setupkvm()`.

26. Difference between `inituvm()` and user address in system manager

- `inituvm()`: This function is used to initialize the user part of the address space for a new process. It allocates one page of physical memory, copies the init executable into that memory, and sets up a PTE [14] for the first page of the user virtual address space.
- User Address: The user address space in xv6 is the range of virtual addresses that a user process can access. For any process, the user memory virtual address (VA) range is from 0 to KERNBASE, where KERNBASE is 0x80000000 (i.e. 2 GB of memory) is available to each process. When a user process requests for memory to build up its user part of the address space, the kernel allocates memory to the user process from a free space list

[14] Page Entry Table

27. What parts of system initialization are exclusive and which are shared

In the xv6 operating system, the initialization process involves several steps that are shared across all processors' kernels. These steps include setting up the kernel and user environments, initializing the scheduler, and starting the first user process.

Kernel Setup: The first kernel, which performs the boot process, enters the main function in the `main.c` file through the `entry.S` code. All system preparation functions called in this function are executed by this kernel.

Other Kernels Setup: Other kernels enter the `mpenter` function through the `entryother.S` code. In this function, 4 functions are also called for preparation.

```
static void
mpenter(void) {
  switchkvm();
  seginit();
  lapicinit();
  mpmain();
}
```

Common Functions: These 4 functions will be common among all kernels. However, one of these functions, named `switchkvm`, is not directly common with the first kernel. This function is called in `mpenter`, while it does not exist in the main function. In fact, the `kvmalloc` function that is called in main is equivalent. The reason that `switchkvm` is shared is because all of them should store the page table created by the first kernel.

```
void
kvmalloc(void)  {
  kpgdir = setupkvm();
  switchkvm();
}
```

User Process Initialization: The initialization of the first user process is a crucial step that involves setting up its address space and starting its execution. But some are exclusive for instance this is only for the first kernel. like:

```c
int main(void) {
    kinit1(end, P2V(4 * 1024 * 1024));         // phys page allocator
    kvmalloc();                                 // kernel page table
    mpinit();                                   // detect other processors
    lapicinit();                                // interrupt controller
    seginit();                                  // segment descriptors
    picinit();                                  // disable pic
    ioapicinit();                               // another interrupt controller
    consoleinit();                              // console hardware
    uartinit();                                 // serial port
    pinit();                                    // process table
    tvinit();                                   // trap vectors
    binit();                                    // buffer cache
    fileinit();                                 // file table
    ideinit();                                  // disk
    startothers();                              // start other processors
    kinit2(P2V(4 * 1024 * 1024), P2V(PHYSTOP)); // must come after startothers()
    userinit();                                 // first user process
    mpmain();                                   // finish this processor's setup
}
```

Some of them are obvious, for instance `startothers` should be executed by the first kernel only. Or disk initialization should be done only once. `mpmain` is also common for all of them since all of the processors' kernels should be setup, this function also calls `scheduler` which handles per-CPU process scheduling.

28. Equivalent of `inicode.S` in linux

https://github.com/torvalds/linux/blob/master/arch/arm/boot/bootp/init.S

Debugging

GDB Overview

There are two kinds of debugging xv6 using GDB.

1. kernel debug
2. user-programs debug

But it must be considered that in two methods, both kernel and user-programs will be executed but debugging is being done only at kernel-level or user-level.

1. Show breakpoints

First, there are several ways to set a breakpoint:

- Line breakpoints: `break <line_number>`
- Function breakpoints: `break <function_namde>`
- File breakpoints: `break <fil_name>:<line_number>`
- Address breakpoints: `break *<address>`
- Conditional breakpoints: `break <location> if <condition>`

To see set breakpoints `info breakpoints` can be used. The information which be provided contains:

```
(gdb) break console.c:558
Breakpoint 1 at 0x80101349: file console.c, line 558.
(gdb) break consputs
Breakpoint 2 at 0x80100970: file console.c, line 216.
(gdb) break exec
Breakpoint 3 at 0x80101570: file exec.c, line 12.
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x80101349 in consoleintr at console.c:558
2       breakpoint     keep y   0x80100970 in consputs at console.c:216
3       breakpoint     keep y   0x80101570 in exec at exec.c:12
```

- Num: Breakpoint number assigned by GDB.
- Type: Type of breakpoint ("breakpoint" or "watchpoint").
- Disp: Breakpoint disposition ("keep" or "delete").
- Enb: Breakpoint enabled state ("y" for enabled or "n" for disabled).
- Address: Memory address where the breakpoint is set.
- What: Location and description of the breakpoint (file name, line number, and function name).

2. Delete a breakpoint

To delete a breakpoint command `delete <breakpoint_number>` can be used. Consider that the breakpoint number can be found with the `info breakpoints` command. Also command `clear <file_name>:<line_number>` can be used if we have insight about breakpoints we set.

Control GDB process

3. Backtrace

GDB always traces and lists the chain of function calls that led to the current function. Command `bt` which is a short form of command `backtrace` shows the call stack of function calls leading to the current point of execution.

As you can see in the image below, `bt` shows a list of stack frames, each function has its own information which is local variables, arguments, return address, etc. The stack frame is organized in a last-in, first-out manner on the call stack, so we can backtrace our calls and find out where the bug occures.

```
(gdb) bt
#0  consoleintr (getc=<optimized out>) at console.c:558
#1  0x0000000a in ?? ()
#2  0x8010c4ac in stack ()
#3  0x80101316 in consputc (c=<optimized out>) at console.c:314
#4  inputputc (c=10 '\n') at console.c:314
#5  consoleintr (getc=<optimized out>) at console.c:554
#6  0x00000000 in ?? ()
```

4. Differences of `print` and `x`

First, let's take a look to help of these operators:

```
(gdb) help print
Print value of expression EXP.
Usage: print [[OPTION]... --] [/FMT] [EXP]

Options:
  -address [on|off]
    Set printing of addresses.

  -array [on|off]
    Set pretty formatting of arrays.

  -array-indexes [on|off]
    Set printing of array indexes.

  -elements NUMBER|unlimited
    Set limit on string chars or array elements to print.
    "unlimited" causes there to be no limit.

  -max-depth NUMBER|unlimited
    Set maximum print depth for nested structures, unions and arrays.
    When structures, unions, or arrays are nested beyond this depth then they
    will be replaced with either '{...}' or '(...)' depending on the language.
    Use "unlimited" to print the complete structure.

  -null-stop [on|off]
    Set printing of char arrays to stop at first null char.

  -object [on|off]
    Set printing of C++ virtual function tables.

  -pretty [on|off]
    Set pretty formatting of structures.

  -raw-values [on|off]
    Set whether to print values in raw form.
    If set, values are printed in raw form, bypassing any
    pretty-printers for that value.

  -repeats NUMBER|unlimited
    Set threshold for repeated print elements.
    "unlimited" causes all elements to be individually printed.

  -static-members [on|off]
    Set printing of C++ static members.
```

```
(gdb) help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
  t(binary), f(float), a(address), i(instruction), c(char), s(string)
  and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format.  If a negative number is specified, memory is
examined backward from the address.

Defaults for format and size letters are those previously used.
Default count is 1.  Default address is following last thing printed
with this command or "print".
```

The print command is used to examine the value of a variable or expression while The x command is used to examine memory contents at a specified address.

In addition, you can print the address of a value using & before values, the output of print is always value but the x command provides more low-level and raw output. It allows you to specify the format in which the memory contents should be displayed. In this case we use x/d to show the value of memory x in decimal.

```
(gdb) print input.e
$1 = 13
(gdb) print &input.e
$2 = (uint *) 0x80110fa8 <input+136>
(gdb) x/d 0x80110fa8
0x80110fa8 <input+136>: 13
```

To see value of a specific register you can use command info registers <register_name>

```
(gdb) info registers eax
eax            0xd                 13
(gdb) info registers ebx
ebx            0xa                 10
```

5. registers and local variable status

EDI and ESI were explained earlier here. Registers and local variables status are explained in other parts as well.

By `info registers` we can see registers state:

```
(gdb) info registers
eax            0x0                 0
ecx            0x0                 0
edx            0x0                 0
ebx            0x82                130
esp            0x8010b500          0x8010b500 <stack+3904>
ebp            0x8010b508          0x8010b508 <stack+3912>
esi            0x80113540          -2146355904
edi            0x80112fa4          -2146357340
eip            0x80103bf5          0x80103bf5 <mycpu+21>
eflags         0x46                [ IOPL=0 ZF PF ]
cs             0x8                 8
ss             0x10                16
ds             0x10                16
es             0x10                16
fs             0x0                 0
gs             0x0                 0
fs_base        0x0                 0
gs_base        0x0                 0
k_gs_base      0x0                 0
cr0            0x80010011          [ PG WP ET PE ]
cr2            0x0                 0
cr3            0x3ff000            [ PDBR=0 PCID=0 ]
cr4            0x10                [ PSE ]
cr8            0x0                 0
efer           0x0                 [ ]
xmm0           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 =
 {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm1           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 =
 {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm2           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 =
 {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm3           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 =
 {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm4           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 =
 {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm5           {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 =
--Type <RET> for more, q to quit, c to continue without paging--c
```

For local variables we can use `info locals`

```
(gdb) info locals
fd = <optimized out>
i = <optimized out>
```

6. input structure

```
struct {
    char buf[INPUT_BUF];
    uint r;     // Read idx
    uint w;     // Write idx
    uint e;     // Edit idx
    uint shift; // number of times cursor has been shifted to left (>= 0)
} input;
```

- `char buf[INPUT_BUF]` is an array of characters which represents our input buffer. It is also considerable that the buffer is implemented in a circular way.
- `uint r` is a index to save index of last character we have read from `buf`

- `uint w` is the index of the first character that is in the buffer but has not been read.
- `uint e` determines the current index of the buffer which is being written to.
- `uint shift` is an added variable which determines the number of times cursor has been shifted to left, which is obviously greater or equal to 0.

To see initial value of input struct, we need to set a breakpoint in `consoleinit` function, where all console-related variables will be initialized.

```
(gdb) b consoleinit
Breakpoint 1 at 0x80101460: file console.c, line 600.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x80101460 <consoleinit>:    endbr32

Thread 1 hit Breakpoint 1, consoleinit () at console.c:600
600     void consoleinit(void) {
(gdb) p input
$1 = {buf = '\000' <repeats 127 times>, r = 0, w = 0, e = 0, shift = 0}
(gdb)
```

At the beginning (after initialization), according to the fact that the input struct is defined as a static struct, all values will be zero. now we add a breakpoint at end of `consoleintr` where changes of entered character have been applied to the input struct:

Note: After we continue debugging, GDB waits until we enter a command in xv6.

We set 2 breakpoints:

1. at `console:556`, which is after receiving any character
2. at `console:558`, which is after receiving `\n`

After each entered character GDB will interrupt qemu, informing us that we hit a breakpoint and determining where the program has been stopped. As we set a breakpoint after reading every character, we can track input struct step by step, see the image below. As you can see, every step a character added to buffer and `input.e` is increased by one.

```
Breakpoint 1 at 0x80101323: file console.c, line 556.
(gdb) break console.c:558
Breakpoint 2 at 0x80101349: file console.c, line 558.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x80101323 <consoleintr+2387>:       cmp    $0xa,%ebx

Thread 1 hit Breakpoint 1, consoleintr (getc=<optimized out>) at console.c:556
556                     if (c == '\n' || c == C('D') || input.e == input.r + INPUT_BUF) {
(gdb) print input
$1 = {buf = "m", '\000' <repeats 126 times>, r = 0, w = 0, e = 1, shift = 0}
(gdb) c
Continuing.
=> 0x80101323 <consoleintr+2387>:       cmp    $0xa,%ebx

Thread 1 hit Breakpoint 1, consoleintr (getc=<optimized out>) at console.c:556
556                     if (c == '\n' || c == C('D') || input.e == input.r + INPUT_BUF) {
(gdb) print input
$2 = {buf = "ma", '\000' <repeats 125 times>, r = 0, w = 0, e = 2, shift = 0}
(gdb) c
Continuing.
=> 0x80101323 <consoleintr+2387>:       cmp    $0xa,%ebx

Thread 1 hit Breakpoint 1, consoleintr (getc=<optimized out>) at console.c:556
556                     if (c == '\n' || c == C('D') || input.e == input.r + INPUT_BUF) {
(gdb) print input
$3 = {buf = "mat", '\000' <repeats 124 times>, r = 0, w = 0, e = 3, shift = 0}
(gdb) c
Continuing.
=> 0x80101323 <consoleintr+2387>:       cmp    $0xa,%ebx

Thread 1 hit Breakpoint 1, consoleintr (getc=<optimized out>) at console.c:556
556                     if (c == '\n' || c == C('D') || input.e == input.r + INPUT_BUF) {
(gdb) print input
$4 = {buf = "mati", '\000' <repeats 123 times>, r = 0, w = 0, e = 4, shift = 0}
(gdb) c
Continuing.
=> 0x80101323 <consoleintr+2387>:       cmp    $0xa,%ebx

Thread 1 hit Breakpoint 1, consoleintr (getc=<optimized out>) at console.c:556
556                     if (c == '\n' || c == C('D') || input.e == input.r + INPUT_BUF) {
(gdb) print input
$5 = {buf = "matin", '\000' <repeats 122 times>, r = 0, w = 0, e = 5, shift = 0}
```

Then we press enter, which adds `\n` to the buffer and the program will hit our second breakpoint, we can see the buffer below. `input.w` is assigned to `input.e` showing that we have entered our command successfully.

```
Thread 1 hit Breakpoint 1, consoleintr (getc=<optimized out>) at console.c:556
556                     if (c == '\n' || c == C('D') || input.e == input.r + INPUT_BUF) {
(gdb) print input
$6 = {buf = "matin\n", '\000' <repeats 121 times>, r = 0, w = 0, e = 6, shift = 0}
(gdb) c
Continuing.
=> 0x80101349 <consoleintr+2425>:       push   $0x80110fa0

Thread 1 hit Breakpoint 2, consoleintr (getc=<optimized out>) at console.c:558
558                         wakeup(&input.r);
(gdb) print input
$7 = {buf = "matin\n", '\000' <repeats 121 times>, r = 0, w = 6, e = 6, shift = 0}
(gdb)
```

After that, we manually interrupt the program to see what happens to `input.r` after the `wakeup` function. `input.r` is equal to `input.w` so it means the entered command has been successfully read by shell, then we can see our command on terminal. We can manually interrupt the program by Ctrl + C, so the program will stop at any point it is at.

```
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
=> 0x80105080 <acquire+48>:     mov    0x8(%ebp),%ebx
0x80105080 in xchg (newval=<optimized out>, addr=<optimized out>) at spinlock.c:28
28          if(holding(lk))
(gdb) print input
$8 = {buf = "matin\n", '\000' <repeats 121 times>, r = 6, w = 6, e = 6, shift = 0}
```

After that we write another command to just show the `input.buf`. buffer won't be cleared after commands, just changing values of `r, w and e` can determine our latest command.

```
(gdb) c
Continuing.
=> 0x80101349 <consoleintr+2425>:       push   $0x80110fa0

Thread 1 hit Breakpoint 2, consoleintr (getc=<optimized out>) at console.c:558
558                         wakeup(&input.r);
(gdb) print input
$9 = {buf = "matin\nsecond command\n", '\000' <repeats 106 times>, r = 6, w = 21, e = 21, shift = 0}
(gdb) 
```

The terminal will look like:

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00



Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
1. Matin Bazrafshan
2. Shahriar Attar
3. Sobhan Alaeddini
$ matin
exec: fail
exec matin failed
$ second command

_
```

Now by re-running our program we want to see if our features work or not. we write a command and shift the cursor to left, see the `input.shift` and terminal below to figure out how the implementation that you see early works.

```
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
The target architecture is assumed to be i386
=> 0x801043d5 <mycpu+21>:       mov     0x801142a0,%esi
0x801043d5 in mycpu () at proc.c:45
45        apicid = lapicid();
(gdb) print input
$1 = {buf = "to test cursor shift", '\000' <repeats 107 times>, r = 0, w = 0, e = 20, shift = 12}
```

Where terminal is:

```
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
1. Matin Bazrafshan
2. Shahriar Attar
3. Sobhan Alaeddini
$ to test cursor shift
```

Now we want to write in the middle of command, pay close attention to `input struct` to see how variables are changed.

```
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
=> 0x801043d5 <mycpu+21>:       mov     0x801142a0,%esi
0x801043d5 in mycpu () at proc.c:45
45        apicid = lapicid();
(gdb) print input
$2 = {buf = "now we update console aftrer cursor shift", '\000' <repeats 86 times>, r = 0, w = 0, e = 41, shift = 12}
```

Now terminal looks like this:

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00



Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
1. Matin Bazrafshan
2. Shahriar Attar
3. Sobhan Alaeddini
$ now we update console aftrer cursor shift
```

After that we simply entered Ctrl + L, to clear the terminal, look at how variables reset but buffer is unchanged.

```
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
=> 0x801043d5 <mycpu+21>:       mov    0x801142a0,%esi
0x801043d5 in mycpu () at proc.c:45
45          apicid = lapicid();
(gdb) print input
$3 = {buf = "now we update console aftrer cursor shift", '\000' <repeats 86 times>, r = 0, w = 0, e = 0, shift = 0}
```

terminal looks like:

```
$ _
```

Debugging assembly code

7. TUI[15]

In the TUI mode in GDB, the screen is split in two sectors and source code will be shown, then we can see source code of where the program has been stopped.

- `layout src` is used to show source code

```
strdiff.c
        if (str1 == 0 || str2 == 0)
            return 0;

        int min_len, max_len, max_str = 0;
        if (strlen(str1) >= strlen(str2)) {
            max_str = 1;
            max_len = strlen(str1);
            min_len = strlen(str2);
        }
        else {
            max_len = strlen(str2);
            min_len = strlen(str1);
        }

        int i;
B+>     for (i = 0; i < min_len; i++) {
            if (str1[i] >= str2[i])
                diff[i] = '0';
            else
                diff[i] = '1';
        }
        for (; i < max_len; i++) {
            if (max_str == 0)
                diff[i] = '1';
            else
                diff[i] = '0';
        }

        diff[i++] = '\n';
        diff[i] = '\0';
        return 1;
    }
```

[15] Text User Interface

- `layout asm` is used to show equivalence assembly code of source code.

```
    0x1bf <strdiff+47>      je     0x270 <strdiff+224>
    0x1c5 <strdiff+53>      sub    $0xc,%esp
    0x1c8 <strdiff+56>      push   %ebx
    0x1c9 <strdiff+57>      call   0x340 <strlen>
    0x1ce <strdiff+62>      mov    %esi,(%esp)
    0x1d1 <strdiff+65>      mov    %eax,%edi
    0x1d3 <strdiff+67>      call   0x340 <strlen>
    0x1d8 <strdiff+72>      add    $0x10,%esp
    0x1db <strdiff+75>      cmp    %eax,%edi
    0x1dd <strdiff+77>      jb     0x280 <strdiff+240>
    0x1e3 <strdiff+83>      sub    $0xc,%esp
    0x1e6 <strdiff+86>      push   %ebx
    0x1e7 <strdiff+87>      call   0x340 <strlen>
    0x1ec <strdiff+92>      mov    %esi,(%esp)
    0x1ef <strdiff+95>      mov    %eax,%edi
    0x1f1 <strdiff+97>      call   0x340 <strlen>
    0x1f6 <strdiff+102>     movl   $0x1,-0x1c(%ebp)
    0x1fd <strdiff+109>     add    $0x10,%esp
B+> 0x200 <strdiff+112>     test   %eax,%eax
    0x202 <strdiff+114>     jle    0x2a8 <strdiff+280>
    0x208 <strdiff+120>     xor    %ecx,%ecx
    0x20a <strdiff+122>     jmp    0x212 <strdiff+130>
    0x20c <strdiff+124>     lea    0x0(%esi,%eiz,1),%esi
    0x210 <strdiff+128>     mov    %edx,%ecx
    0x212 <strdiff+130>     movzbl (%esi,%ecx,1),%edx
    0x216 <strdiff+134>     cmp    %dl,(%ebx,%ecx,1)
    0x219 <strdiff+137>     setl   %dl
    0x21c <strdiff+140>     add    $0x30,%edx
    0x21f <strdiff+143>     mov    %dl,0xdf0(%ecx)
    0x225 <strdiff+149>     lea    0x1(%ecx),%edx
    0x228 <strdiff+152>     cmp    %edx,%eax
    0x22a <strdiff+154>     jne    0x210 <strdiff+128>
```

- `layout split` shows both assembly and source code of program:

```
strdiff.c
B+>48           for (i = 0; i < min_len; i++) {
   49               if (str1[i] >= str2[i])
   50                   diff[i] = '0';
   51               else
   52                   diff[i] = '1';
   53           }
   54           for (; i < max_len; i++) {
   55               if (max_str == 0)
   56                   diff[i] = '1';
   57               else
   58                   diff[i] = '0';
   59           }
   60
   61           diff[i++] = '\n';
   62           diff[i] = '\0';

B+>0x200 <strdiff+112>     test   %eax,%eax
   0x202 <strdiff+114>     jle    0x2a8 <strdiff+280>
   0x208 <strdiff+120>     xor    %ecx,%ecx
   0x20a <strdiff+122>     jmp    0x212 <strdiff+130>
   0x20c <strdiff+124>     lea    0x0(%esi,%eiz,1),%esi
   0x210 <strdiff+128>     mov    %edx,%ecx
   0x212 <strdiff+130>     movzbl (%esi,%ecx,1),%edx
   0x216 <strdiff+134>     cmp    %dl,(%ebx,%ecx,1)
   0x219 <strdiff+137>     setl   %dl
   0x21c <strdiff+140>     add    $0x30,%edx
   0x21f <strdiff+143>     mov    %dl,0xdf0(%ecx)
   0x225 <strdiff+149>     lea    0x1(%ecx),%edx
   0x228 <strdiff+152>     cmp    %edx,%eax
   0x22a <strdiff+154>     jne    0x210 <strdiff+128>
   0x22c <strdiff+156>     add    $0x2,%ecx
   0x22f <strdiff+159>     cmp    %edx,%edi
```

8. Move between functions in frame stack

- command up is used to jump out of the function to where the function is called from. It has the same functionality as step out.
- command down is used to jump to the next function in the frame stack. It has the same functionality as stepin.

for example after we call up in the `strdiff` function, we can see:

```
        if (fd < 0) {
            printf(2, "Error happens when trying making file!\n");
            exit();
        }

>       if (strdiff(argv[1], argv[2]) == 0) {
            printf(2, "String must only include alphabetical characters!\n");
            exit();
        }

        write(fd, diff, strlen(diff));
        close(fd);

        exit();
    }
```

Linux (optional):

After we installed Linux Ubuntu 22.04 we checked our system. Core is 5.15.0 using `uname-a`:

```
make: *** [Makefile:733: .config] Error 1
linux-5.15.136: uname -a
Linux sobhan-virtual-machine 5.15.0-83-generic #92~20.04.1-Ubuntu SMP Mon Aug 21 14:00:49 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
linux-5.15.136:
```

To bring the new kernel version closer to the previous one, we are using kernel 5.19.136. We utilize the `make defconfig` command for a smaller size and faster compilation during the configuration process. After replacing the kernel and booting it using "qemu" , we observe the presence of the new version by using the `uname -a` command. Therefore, the kernel has been changed:

```
(initramfs) uname -a
Linux (none) 5.15.136 #2 SMP Sun Oct 22 06:03:19 EDT 2023 x86_64 GNU/Linux
(initramfs)
```

To incorporate the `dmesg` feature into our code, we made changes to the "main.c" file within the initialization file of our Linux version. Here is the revised format of the message:

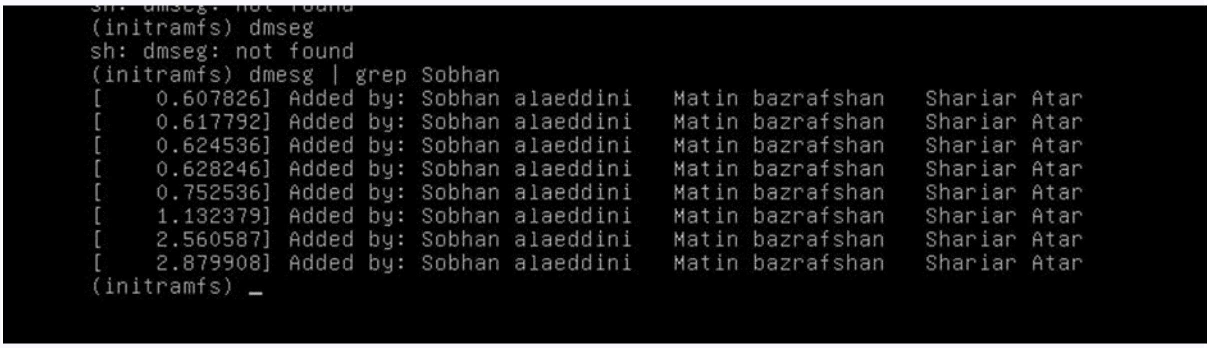
We modified the `main.c` file in the init file of our Linux version to add the 'dmesg' feature. The changes made are as follows:

```
static void __init do_initcall_level(int level, char
*command_line)
{
	initcall_entry_t *fn;

	parse_args(initcall_level_names[level],
		   command_line, __start___param,
		   __stop___param - __start___param,
		   level, level,
		   NULL, ignore_unknown_bootoption);

	trace_initcall_level(initcall_level_names[level]);
	for (fn = initcall_levels[level]; fn <
initcall_levels[level+1]; fn++)
		do_one_initcall(initcall_from_entry(fn));
const char* yourName = "1.Sobhan Alaeddini  2.Shariar Attar
3.Matin Bazrafshan";
printk(KERN_INFO "Added by: %s\n", yourName);
}
```

After writing `dmesg` it shows our name and work correctly:



Booting process:

After we make our file we want to boot it on ‘qemu’ with 1G of ram:

```
qemu-system-x86_64 -kernel <kernel_image> -m 1G
```

Now we want to know what a kernel_image is: In general, a kernel image refers to the binary file that contains the operating system's kernel code. The kernel is the core component of an operating system that manages system resources, provides essential services, and acts as an interface between software and hardware.

Acknowledgements

We would like to begin by expressing our deepest gratitude to all the collaborators who have contributed to this report. Their dedication and hard work have been instrumental in completing all the practical and mandatory parts of this project. Their expertise and commitment have been invaluable, and this work would not have been possible without their contributions.

For some of the additional sections (not mandatory) and questions that were entirely research-based, we sought assistance from various resources. We utilized ChatGPT, a state-of-the-art language model, for generating creative content and providing insightful suggestions. We also extensively used the internet for gathering information and understanding different perspectives.

Furthermore, we referred to the xv6 reference extensively to gain a deeper understanding of the concepts and to ensure the accuracy of our work. The xv6 reference has been an excellent resource, providing us with detailed explanations and valuable insights.

References

In the spirit of maintaining academic integrity, we would like to note that all resources, including the xv6 reference, were used “as is”. We have made sure to respect the original work and have not altered or misrepresented any information. All sources have been appropriately cited, giving credit to the original authors. We believe in the importance of academic honesty and strive to uphold these values in all our work.