COMP604 - Operating Systems

Unofficial course manual

Written by Zach Barrett Last updated: 2024-12-09

Table of contents

1. Introduction

This document is a collection of my notes and knowledge related to the class “Operating Systems” at AUT, written for someone taking this class. While largely intended as a way of writting down everything I know for my own benefit, my hope is that it will be of use for you as well.

1.1. Additional resources

This class recommends the use of the OSTEP (Operating Systems, Three Easy Pieces) textbook. It can be found here: OSTEP This is an excellent resource, and I highly recommend using this.

2. Class Overview

2.1. Assessment Overview

There are three assignments in this class, with assignment two being the longest and most difficult.

Assessment Item	Weighting
Assignment One	30%
Assignment Two	40%
Assignment Three	30%

2.1.1. Assignment One

This assignment covered shell scripting, C programming, and a simple xv6 system call. Easiest by far, do not use your extension here. When I took it, this assignment had 3 questions.

2.1.2. Assignment Two

This assignment covered scheduling algorithms, processes, and more advanced system calls. This is the longest assignment, using your extension here is fine. When I took it, this assignment had 5 questions. All programming questions were done in xv6.

2.1.3. Assignment Three

This assignment covered semaphores, inodes, RAID, storage in xv6. This assignment was harder than assignment one but easier than assignment two. If you can save your extension for this assignment then great, but don’t feel bad about spending it on assignment two.

2.2. Software tools

The later portion of this course makes use of the XV6 operating system, which is a simple “toy” operating system which is easy to modify.

Because XV6 runs on a different architecture than most computers, it can only be run using a virtual machine. Thankfully, a free and easy-to-use virtual machine exists called QEMU. The next sections will show you how to install this software stack.

If you are able to get access to a Linux or Mac computer, doing so will make your life much easier as you will be able to run QEMU directly and not through another virtual machine (Unlike Windows, where you will need to use WSL).

If you have to use WSL, I highly recommend installing Visual Studio Code and the WSL extension for it, allowing you to edit the xv6 files directly through the editor instead of having to either copy them in/out or having to use a command line editor like nano or vim. (Although I also recommend you learn how to use vim keybindings.)

This information is a combination of the “Software setup” section on Canvas and my own experiences.

2.2.1. Mac setup

1. To compile the software tools necessary to run XV6, you must first install Xcode.

   ```
   xcode-select --install
   ```

   This will take some time.

2. Next, ensure the Homebrew package manager is installed. If it is not, install with the following command:

   ```
   /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
   ```

   (If you don't already use Homebrew, you should. It's extremely useful.)

3. Then, use Homebrew to install the RISC-V compiler toolchain:

   ```
   brew tap riscv/riscv
   brew install riscv-tools
   ```

   With this step, sometimes people (me, I had this issue) have issues with compiling the RISC-V toolchain. If that occurs try changing your OSX version to OSX Sonoma, as there is a precompiled binary available for that version. Once that's done try running those commands again, Homebrew will handle it for you.

4. Next you must add the toolchain directory to your "path", which is where your computer looks for programs if they are not in the current directory. This can be done using any text editor. To do so, you must open a file called `.zshrc` (This is the configuration file for the default shell on Mac, if you have changed the shell I assume you know which **other** file to change instead.)

   Then append one of the following lines to the bottom of the file depending on your situation

   ```
   # Use the below line on M series Macs
   export PATH=$PATH:/usr/local/opt/homebrew/Cellar/riscv-gnu-toolchain/master/bin

   # Use this one for Intel series Macs
   export PATH=$PATH:/usr/local/Cellar/riscv-gnu-toolchain/master/bin

   # If you are on an M series Mac and you get an error message at the end of
   # setup saying the computer can't find the compiler toolchain or something,
   # replace the line you added with this one:
   export PATH=$PATH:/opt/homebrew/Cellar/riscv-gnu-toolchain/main.reinstall/bin
   ```

5. Once that is done, tell zsh to use the new configuration file:

   ```
   source ~/.zshrc
   ```

6. Finally, install QEMU.

   ```
   brew install qemu
   ```

   If you have issues with this step ensure that the path was set correctly.

2.2.2. Ubuntu setup

Getting this running is very easy on Ubuntu. Note that the university computers with WSL enabled also have Ubuntu installed.

To setup QEMU on Ubuntu, simply run the following commands:

```
sudo apt update && sudo apt upgrade
sudo apt install build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

If that throws an error, try separating all of the packages into separate lines in a text file (Let's call it packs.txt), and running the following command:

```
cat packs.txt | xargs sudo apt install
```

(Learn about the xargs command by the way, very useful!)

2.2.3. Windows setup

You will need to use WSL (Windows Subsystem for Linux) as Windows cannot run QEMU directly.

1. Start by opening a command prompt and running the following command:

   ```
   wsl --install -d ubuntu-22.04
   ```

   This will install WSL and Ubuntu.
2. Reboot your computer, open BIOS and ensure that hardware virtualization is enabled. (look up how to do this, it varies by motherboard manufacturer)
3. Once your computer has rebooted, open a command prompt and run:

   ```
   ubuntu
   ```

   This will open an Ubuntu command prompt. From here setup proceeds the same as in the Ubuntu setup section.

   Once you have completed that, I recommend setting up an editor capable of opening and editing files in WSL. This can be done using Visual Studio Code and an extension. Here is the link to that extension

3. What you need to know about C programming

Almost all of the programming in this class will be done using C. In addition to the basic C syntax you likely learned in earlier classes, you will also be using more advanced concepts.

The following is an overview of the two main concepts you will need to revise.

3.1. Header files

C header files (ending with .h) contain function declarations and macro definitions that can be shared across multiple source files. This means you can write a function once in a .c file, declare it in a header file, and then use it in any other file that includes that header.

Making your own header file is extremely simple - just create a new file ending with .h and put your function declarations in there. These declarations are all that is needed for C to know that these functions exist.

This is the reason that we include stdio.h at the top of programs, it tells C what functions are available in the standard input/output library (such as printf).

Generally each C source file should have an associated header file with the same name, just with .h instead of .c.

Additionally, a distinction must be made between local headers and system headers. local headers are specified by using their path between quotes (“myheader.h”), and system headers are specified with angle brackets ().

Here is an example:

```
// In myprogram.c
#include <stdio.h>      // System header
#include "myheader.h"   // Local header
```

The difference between the two styles is that when the compiler searches for a header file, it looks in different places depending on which style is used.

When using angle brackets (<>), the compiler looks in standard system directories first (like /usr/include).

When using quotes (“”), the compiler first looks in the current directory, then in the specified include paths, and finally in the system directories.

This is why we use quotes for our own header files (which are usually in the same directory as our source files) and angle brackets for system headers (which are installed in system locations).

3.1.1. XV6 Headers

XV6 has several header files, each containing functions related to a specific purpose:

- user.h for user-level function calls
- proc.h for the process structure and scheduler
- defs.h for kernel internal functions
- types.h for data type definitions

When making system calls in XV6, you'll need to include user.h in your user programs to access syscall functions, and both defs.h and proc.h for kernel functions.

One key difference from regular C is that XV6 defines its own types in types.h rather than using the standard C types, so you'll use uint instead of unsigned int, etc.

3.2. Pointers

Pointers are variables that store memory addresses of other variables. These are used extensively in operating systems, so I highly recommend studying them.

A pointer is declared by adding an asterisk (*) before the variable name:

```
int* ptr;    // Declares a pointer to an integer
char* str;   // Declares a pointer to a character
```

To get the address of a variable, use the ampersand (&) operator:

```
int x = 5;
int* ptr = &x;  // ptr now holds the memory address of x
```

To access the value that a pointer points to (dereferencing), use the asterisk (*):

```
int value = *ptr;  // value is now 5
```

Structures can also be accessed through pointers. Here's how:

```
struct Point {
    int x;
    int y;
};

struct Point p = {10, 20};
struct Point* ptr = &p;

// These two lines do the same thing:
printf("%d", (*ptr).x);
printf("%d", ptr->x);    // Easier to use!
```

The arrow operator (->) is shorthand for dereferencing a pointer and accessing a structure member. It's much more readable than the alternative.

Null pointers (pointers that don't point to anything) are declared as:

```
int* ptr = NULL;
```

Always check if a pointer is NULL before dereferencing it to avoid segmentation faults.

Common pointer mistakes to avoid:
• Dereferencing NULL pointers
• Using uninitialized pointers
• Not freeing allocated memory (memory leaks)
• Writing beyond array bounds
• Using dangling pointers (pointing to freed memory)

Pointers and arrays are closely related in C. An array name is actually a pointer to its first element. This is why array indexing and pointer arithmetic are interchangeable:

```
int arr[5] = {1, 2, 3, 4, 5};
int* ptr = arr;      // ptr points to first element
printf("%d", ptr[2]); // Prints 3
printf("%d", *(ptr + 2)); // Also prints 3
```

3.3. Memory management

In C, dynamic memory allocation allows you to allocate memory at runtime rather than compile time. This is done using functions from the stdlib.h library:

```
// Allocate memory for one integer
int* ptr = (int*)malloc(sizeof(int));

// Allocate memory for an array of 10 integers
int* arr = (int*)malloc(10 * sizeof(int));

// Always check if allocation succeeded
if (ptr == NULL || arr == NULL) {
    // Handle allocation failure
    return -1;
}

// Use the memory
*ptr = 42;
arr[0] = 1;

// Free the memory when done
free(ptr);
free(arr);
```

The malloc() function returns a void pointer that must be cast to the appropriate type. malloc() allocates the specified number of bytes and returns a pointer to the first byte.

There are other memory allocation functions:
• calloc(): Allocates and zeros memory
• realloc(): Resizes previously allocated memory
• free(): Releases allocated memory

Memory that is dynamically allocated exists on the heap rather than the stack. Stack memory is automatically managed and freed when variables go out of scope, while heap memory must be manually freed using free().

Failing to free allocated memory results in memory leaks - memory that remains allocated but can no longer be accessed by the program. This is especially problematic in long-running programs like operating systems.

A few important rules for dynamic memory:
• Always check if malloc() returns NULL
• Only free memory once
• Only free memory allocated with malloc/calloc/realloc
• Keep track of all allocated memory
• Set pointers to NULL after freeing them

3.3.1. Dynamic memory management in xv6

Memory management in xv6 is almost identical to normal C, except memory allocation is dont using the `malloc()` system call instead of the stdlib function.

4. Command Line Utilities

4.1. Basic utilities

These are simple utilities with short descriptions.

4.1.1. cd

Short for ‘change directory’, this utility allows you to move between directories.

Example usage:

```
cd newDirectory/
```

4.1.2. ls

Short for ‘list’, lists the files in a directory. Use the command line flag `-a` (short for ‘all’), to show hidden files and directories.

```
ls
# Or:
ls directoryYouWantToSeeTheContentsOf/
```

4.1.3. man

Short for ‘manual’, displays the manual page for a command. This is **extremely** useful, use this command often.

Example usage:

```
man ls
```

4.1.4. cat

Short for ‘concatenate’, was designed to be used for combining two text files, but it allows us to print the contents of a file to the terminal.

Example usage:

```
cat file.txt
```

4.1.5. pwd

Short for ‘print working directory’, displays the current directory you are in.

Example usage:

```
pwd
```

4.1.6. mkdir

Short for ‘make directory’, creates a new directory.

Example usage:

```
mkdir newDirectory
```

You can also create multiple nested directories at once using the `-p` flag:

```
mkdir -p path/to/new/directory
```

4.1.7. chmod

Short for ‘change mode’, changes the permissions of a file or directory.

Example usage:

```
chmod 755 file.txt
```

The permissions are represented by a three-digit number:
• First digit: owner permissions
• Second digit: group permissions
• Third digit: everyone else’s permissions

Each digit is the sum of:
• 4 (read)
• 2 (write)
• 1 (execute)

So 755 means:
• Owner can read (4), write (2), and execute (1): 4+2+1 = 7
• Group can read (4) and execute (1): 4+0+1 = 5
• Others can read (4) and execute (1): 4+0+1 = 5

you can also quickly make a file executable by running:

```
chmod +x file.sh
```

4.1.8. mv

Short for ‘move’, moves files or directories from one location to another. It can also be used to rename files.

Example usage:

```
# Move a file
mv source.txt destination.txt

# Move a file to a directory
mv file.txt directory/

# Move multiple files to a directory
mv file1.txt file2.txt directory/

# Rename a file
mv oldname.txt newname.txt
```

The mv command will overwrite the destination file if it exists. Use the `-i` flag to make mv prompt before overwriting.

4.1.9. cp

Short for ‘copy’, copies files or directories from one location to another.

Example usage:

```
# Copy a file
cp source.txt destination.txt

# Copy a directory and its contents (recursive)
cp -r sourceDir/ destinationDir/

# Copy while preserving permissions and timestamps
cp -p source.txt destination.txt
```

Unlike mv, cp creates a duplicate of the file, leaving the original intact.

4.2. tar

This command creates an archive from either a list of files or a directory. You will be using this to turn your `xv6-riscv` directory into a file ready for submission.

This is a remarkably complicated command to use, but here are some example usages:

```
# Create a tar archive from a directory
tar -czvf myarchive.tar.gz mydirectory/

# Extract the contents of a tar archive to current directory
tar -xzvf myarchive.tar.gz

# As used in COMP604:
tar -czvf myname-12345678-assX.tar.gz xv6-riscv/
```

4.3. grep

Grep allows you to search for strings of text (or more generally, regexes) in either a file or a stream.

Example usage:

```
# Search for a specific string in a file
grep "string_to_search" filename.txt

# Search for a string in all files in the current directory
grep "string_to_search" *

# Search for a string recursively in all files in current directory and subdirectories
grep -r "string_to_search" .
```

4.4. nano/vim

Both of these are command line text editors. Nano is much easier to use but is more limited. For most of you Nano is the best option. You can use it by running something like: `nano fileToEdit.txt`. To exit, use `ctrl-x`.

Vim is far more complex, but is **absolutely** worth learning how to use. If you learn how to use it well, Vim is faster, easier to use, and more powerful. To open vim, use `vim fileToEdit.txt`. To exit, enter the key combination: `:qa!`, which stands for “quit all, force”.

Either of these is worth knowing how to use, as they are preinstalled on essentially all Linux installs. You can use either of these text editors to edit files inside the Ubuntu virtual machine without the use of an external text editor.

4.5. Pipes and redirection

As a quick aside, one of the powerful things about the command line is the ability to combine commands. This can be done with pipes or redirection. Pipes allow you to take the output of one command and use it as the input of another command.

For example, if you want to find a specific process:

```
ps aux | grep "firefox"
```

This takes the output of ps aux (which shows all running processes) and searches through it for "firefox".

Redirection allows you to send the output of a command to a file instead of the terminal:

```
# Save output to a file (overwrites existing content)
echo "Hello" > file.txt

# Append output to a file
echo "World" >> file.txt

# Redirect both standard output and error messages
command > output.txt 2>&1
```

4.6. Useful utilities you might want to install

4.6.1. tealdear

Tealdear is an implementation of the `tldr` command available on most platforms. It is very similar to the man command, but with a simplified output similar to your average "How to use X" cheatsheet.

4.6.2. neovim

Newer implementation of vim with more features and compatibility with plugins. Worth learning how to use. If you want to get neovim set up quickly, look into kickstart.nvim

4.6.3. fzf

Short for "fuzzy find", essentially an interactive version of grep.

4.6.4. ripgrep

A faster implementation of grep.

5. XV6 for Dummies

The most recent version of xv6 can be found on this GitHub page: xv6-riscv

5.1. General structure

Like most Unix operating systems, xv6 is split into user and kernel space.

Almost all of the operation of xv6 occurs in kernel space, which has elevated privileges to user space.

The distinction is made clear by the fact that kernel programs are kept in the ‘kernel’ folder, and user programs are kept in the ‘user’ folder. Additionally, on the same level as either of these folders is the ‘Makefile’, which performs the startup sequence for xv6. The structure looks something like this:

```
xv6-riscv/
    ├─ Makefile
    ├─ user/
    │  ├─ User programs go here
    ├─ kernel/
         ├─ Kernel programs go here
```

5.2. System Calls

The only method of communicating between programs in kernel and user space is through predefined “System Calls”, which are what you will be spending most of your time creating.

5.2.1. Adding a new system call

Adding a new system call is a complex process.

The following steps need to be performed to add a new system call:

1. In `kernel/syscall.h`, add the system call number. This file contains all of the numbers used by the system calls.

   ```
   #define SYS_examplesyscall 22
   ```

2. In `kernel/syscall.c`, include the prototype for the system call. Near the top of `syscall.h`, add:

   ```
   extern uint64 sys_examplesyscall(void);
   ```

   Then, near the bottom add an entry to the array of function pointers:

   ```
   [SYS_examplesyscall] sys_examplesyscall,
   ```

3. In `kernel/sysproc.c`, implement the system call.

   ```
   uint64
   sys_examplesyscall(void) {
       // Implementation goes here
       return 0;
   }
   ```

4. In `kernel/defs.h`, add a prototype for the system call.

   ```
   uint64 sys_examplesyscall(void);
   ```

5. In `user/usys.pl`, add an entry for your syscall at the bottom:

   ```
   entry("examplesyscall");
   ```

6. Finally, in `user/user.h`, add a prototype of the new system call:

```
int fork(void);
// All other system calls
int examplesyscall(void);
```

For this example I used ‘examplesyscall’ as an example name but you can replace this with whatever name you want to give it.

After completing these steps, you will be able to call this system call from a user program by including the `user.h` header file and calling the function as per usual.

5.3. User Programs

5.3.1. Adding a new user program

To add a new user program to xv6, you must first add the C program under the `user/` directory. This program is essentially the same as any other C program, except it must use `exit(0)` instead of `return 0;`

After doing so, you must modify the Makefile. Starting on line 125 is a section that looks like this:

```
UPROGS=\
  $U/_cat\
  $U/_echo\
  $U/_forktest\
  $U/_grep\
  $U/_init\
  $U/_kill\
  $U/_ln\
  $U/_ls\
  $U/_mkdir\
  $U/_rm\
  $U/_sh\
  $U/_stressfs\
  $U/_usertests\
  $U/_grind\
  $U/_wc\
  $U/_zombie\
```

Add the name of your user program in the same format as the other programs here (with an underscore in front and without the file extension.) It is best to do so at the bottom, to make it easier to find quickly.

5.4. Files of interest

These are the most commonly modified files in xv6:

1. In kernel space:
   - `kernel/proc.c` - Process related functions
   - `kernel/syscall.c` - System call definitions and setup
   - `kernel/sysproc.c` - System call implementations
   - `kernel/defs.h` - Function prototypes
   - `kernel/syscall.h` - System call numbers
   - `kernel/param.h` - System parameters
   - `kernel/proc.h` - Process structure definitions
2. In user space:
   - `user/user.h` - User-space system call declarations
   - `user/usys.pl` - System call stub generator
   - `user/*.c` - Your user programs

The files you will most commonly be editing are:
- `sysproc.c` for implementing system calls
- `proc.c` for process management

5.5. Using xv6

5.5.1. XV6 Setup

Assuming you have run through the earlier software setup section, you should be able to run the command: `make qemu` from within the xv6-riscv directory and be able to start it without issue. However, on MacOS you may need to modify the Makefile to run it successfully. To do so, find line 59 and remove `-Werror` from it.

The region in question looks like this:

```
CC = $(TOOLPREFIX)gcc
AS = $(TOOLPREFIX)gas
LD = $(TOOLPREFIX)ld
OBJCOPY = $(TOOLPREFIX)objcopy
OBJDUMP = $(TOOLPREFIX)objdump

# Modify the below line.
CFLAGS = -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2
CFLAGS += -MD
CFLAGS += -mcmodel=medany
```

Doing so disables strict error checking (Which could be worth doing anyway!). Once you have done so you should make a copy of this version of xv6 for future use, instead of having to repeat this step.

Additionally, while the xv6 download is approximately 18 megabytes, almost all of that is the .git folder (responsible for version tracking using git), removing this makes the whole folder some number of kilobytes. This means that the whole directory can be sent through most messaging apps.

To remove this directory, run:

```
rm -rf ./.git
```

from inside the `xv6-riscv` directory.

5.5.2. Using xv6

The xv6 shell behaves very similarly to a typical shell session. To run a user program, simply type its name and press enter.

You can list all of the running programs by pressing `ctrl-P`

To exit xv6, press `ctrl-a`, release those two keys and then press `x`.

5.5.3. Preparing an xv6 instance for submission

Generally there are two steps for submitting an xv6 file.

1. Clean the xv6 directory
   Run:

   ```
   make clean
   ```

   from inside the 'xv6-riscv/' directory.
   This removes all compiled binaries.

2. Use tar to turn the directory into an archive file.
   This can be done by running something like:

   ```
   tar czf Q1-12345678.tar.gz ./xv6-riscv
   ```

   from **above** the `xv6-riscv` directory. Substitute with the specific question and your student ID sequence.

All of these steps can be automated by writing your own makefile for the assignment. Learn how to do this, it will save you hours of recompiling, cleaning, and trying to remember tar commands.

6. Miscellaneous

6.1. Other useful tricks

6.1.1. Makefiles

As mentioned earlier, makefiles can be used to automate repetitive tasks for this class. This is the makefile I used for assignment 3

```
OUTDIR = "FinalSubmission"
STUDENTID = ""

all: q1 q2 q3 q4
    tar -czf ./Assign-$(STUDENTID).tar.gz ./$(OUTDIR)

q1:
    echo "Question One"
    rm -rf ./$(OUTDIR)/Question_1
    mkdir -p ./$(OUTDIR)/Question_1
    @$(MAKE) -C ./Question_1/xv6-riscv clean
    tar -czf ./$(OUTDIR)/Question_1/Q1-$(STUDENTID).tar.gz ./Question_1/xv6-riscv

q2:
    echo "Question Two"
    mkdir -p ./$(OUTDIR)/Question_2
    typst compile ./Question_2/Q2-$(STUDENTID).typst ./$(OUTDIR)/Question_2/Q2-$(STUDENTID).pdf

q3:
    echo "Question Three"
    @mkdir -p ./$(OUTDIR)/Question_3
    typst compile ./Question_3/Q3-$(STUDENTID).typst ./$(OUTDIR)/Question_3/Q3-$(STUDENTID).pdf

q4:
    echo "Question Four"
    rm -rf ./$(OUTDIR)/Question_4
    mkdir -p ./$(OUTDIR)/Question_4
    @$(MAKE) -C ./Question_4/xv6-riscv clean
    tar -czf ./$(OUTDIR)/Question_4/Q4-$(STUDENTID).tar.gz ./Question_4/xv6-riscv

clean:
    rm -rf ./$(OUTDIR)
    rm -rf ./Assign3-$(STUDENTID).tar.gz
```

This allows me to clean and archive my xv6 directories, and compile my typst documents (typst is a great typesetting system I used for my assignments and this document!).

You can probably take this one and modify it, but I recommend learning how to write them yourself.

6.1.2. Git version management

Git is an extremely useful tool for this class, particularly when working on multiple assignments at once. It allows you to keep track of changes to your code and switch between different versions easily.

Here is a good place to start learning how to use git: Git tutorial

Otherwise, here are some basic commands:

```
# Initialize a git repository
git init

# Add files to be tracked
git add .

# Commit changes
git commit -m "message"

# Add a remote repository
git remote add origin https://github.com/username/repository.git

# Push to remote repository
git push
```

You should use a remote service like github to back up your code, this makes it **much** easier to recover from some disaster or work across multiple computers.

I once heard that you should make backups at least as often as it would take you to redo your work. This advice has saved me days of time.

6.1.3. Documentation

While it can take time and effort, it is best to keep a log of every change you make to xv6.

Some things to document:
1. What files you modified
2. What changes you made to each file
3. Why you made those changes
4. Any issues you encountered
5. How you fixed those issues
6. What testing you performed

By doing this, you make it much easier to:
1. Remember how you did something
2. Fix issues that come up later
3. Explain your changes to others
4. Prove that you did the work
5. Learn from your mistakes
6. Build on your previous work

Even if you think you'll remember how you did something, you probably won't. Documentation is a gift to your future self.