A WEB-BASED SYSTEM PROGRAMMING LEARNING ENVIRONMENT

BY

NEELABH S. GUPTA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Computer Engineering
in the College of Engineering of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Advisers:

Dr. Roy H. Campbell
Dr. Lawrence C. Angrave

ABSTRACT

This thesis introduces a web application designed for students learning system programming. The tool developed supports compiling and running C programs right inside the browser (made possible by a full-featured Linux-based virtual machine running purely client-side), a full-featured editor designed for beginners writing C programs, ability to search the Linux Man pages, and more. Short video lectures and exercises are also available which introduce learners to the C programming language and system programming concepts. The application has been used successfully by more than 400 students for two semesters at the University of Illinois.

In this thesis, the motivation behind developing this application is discussed, along with its features and possible use cases. A thorough walk-through of the user interface is given, followed by elaborate details of the challenges, design, architecture and implementation of the application. The thesis also briefly analyses the performance and usage of the application.

Subject Keywords: System programming; Computer science education; Programming environments; Online learning; Courseware; C programming in the browser; Linux in a browser; Web application architecture; Web design; Open source software

To my family, for their love and support.

ACKNOWLEDGMENTS

I would like to thank Professor Lawrence Angrave for coming up with the initial idea for the project, and for giving me the opportunity to work on it. The project and the thesis would not have been possible without his work, mentorship, guidance, and feedback.

I would like to thank Professor Roy Campbell for providing guidance and feedback on the project as well as on the thesis. His ideas and advice have been indispensable in the completion of the project and this thesis.

In the Fall 2014 semester, this project became part of the Department of Computer Science Senior Projects course [1] at the University, where a team of seven students (including me) worked on the project for two semesters (Fall 2014 – Spring 2015). I would like to thank the course Instructor, Professor Michael Woodley, Project Liaison Professor Angrave, Teaching Assistants Terence Nip and Xiaodan Zhang (Sally), and the team of students who worked on the project — Anant Singh, Eric Ahn, Joseph Tran, Keagan McClelland, Scott Walters, and Siddharth Seth.

My grateful thanks are also extended to Terry Peterson in the ECE Advising office for her help and support on the thesis and to Jamie Hutchinson in the ECE Editorial Services office for providing quick but thorough feedback on this thesis.

Last but not least, I would like to thank my family and friends for their love and support.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND SYMBOLS vi

GLOSSARY OF TERMS viii

CHAPTER 1 INTRODUCTION 1
- 1.1 System Programming Instruction 2
- 1.2 Challenges in Teaching System Programming 3
- 1.3 The Project and Its Advantages 4

CHAPTER 2 FUNCTIONALITY AND USAGE 7
- 2.1 Lessons Page 8
- 2.2 Lecture Video Page 8
- 2.3 Play Activity Page 10
- 2.4 Playground 11

CHAPTER 3 DESIGN AND IMPLEMENTATION 23
- 3.1 Design Decisions 23
- 3.2 Organization of the Project 24
- 3.3 Deployment Architecture 27
- 3.4 Application Architecture 29
- 3.5 Implementation of the “Compile and Run” Flow 32

CHAPTER 4 RESULTS AND ANALYSIS 43
- 4.1 Jor1k Performance Comparison 43
- 4.2 File Size Statistics 45
- 4.3 Usage and Adoption 48

CHAPTER 5 IMPROVEMENTS AND SUGGESTIONS 51
- 5.1 Automatic Grading of Student Code 51
- 5.2 Event Tracking and Analytics 52

CHAPTER 6 CONCLUSION 53

REFERENCES 54

LIST OF ABBREVIATIONS AND SYMBOLS

⌘	The Command key present on Mac keyboards
Ctrl or ctrl	The Control key present on most keyboards
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CDN	Content Delivery Network
CS	Computer Science
CSS	Cascading Style Sheets
DOM	Document Object Model
GB	gigabyte; 1 GB = 1000 MB
GCC	The GNU Compiler Collection, a compiler system produced by the GNU Project supporting various programming languages. This system includes a compiler for the C programming language. The term GCC will be used in this thesis to refer to the C compiler.
gcc	The GCC C compiler command-line executable program
GHz	gigahertz
GiB	gibibyte; 1 GiB = 1024 MiB
GNU	GNU is a collection of free and open source software. GNU is a recursive acronym for "GNU's Not Unix!"
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
ISA	Instruction Set Architecture
JS	JavaScript
JSON	JavaScript Object Notation

KB	kilobyte; 1 KB = 1000 bytes
KiB	kibibyte; 1 KiB = 1024 bytes
KIPS	Thousand instructions per second
MB	megabyte; 1 MB = 1000 KB
MiB	mebibyte; 1 MiB = 1024 KiB
MIPS	Million instructions per second
MOOC	Massive Open Online Course
MP	Machine Problem, a term used for programming assignments at UIUC
OR1K	OpenRISC 1000
POSIX	Portable Operating System Interface
RISC	Reduced Instruction Set Computing
SSH	Secure Shell
`tty`N	The N^{th} virtual terminal of a Linux system
UART	Universal Asynchronous Receiver/Transmitter
UI	User Interface
UIUC	University of Illinois at Urbana-Champaign
URL	Uniform Resource Locator
VM	Virtual Machine
webapp	Web Application
XHR/xhr	XMLHttpRequest, a browser API used by AJAX techniques
XML	Extensible Markup Language

GLOSSARY OF TERMS

Back-End A remote server capable of running application code (as opposed to just serving files statically)

Branch A parallel version of the main line of development in a repository, that is, the default branch (usually `master`)

Client-Side The part of a web application running on the front-end

Fork (of a Repository) A copy of a repository

Front-End The user's web browser

GitHub Organization A central place for GitHub repositories owned by a single group or company

GNU/Linux A distribution/version of the GNU operating system using the Linux kernel

Linux Based on the context, this term can refer either to the Linux kernel, or to a complete GNU/Linux operating system

Linux kernel A popular, Unix-like computer operating system kernel

Repository A set of files and associated metadata contained in a version control system, such as Git

Server-Side The part of a web application running on the back-end

Single-Page Application A web application/site consisting of a single web page, such that actions like navigation, fetching of data, form submissions, etc., are performed without reloading the page

Wiki A website that allows direct collaborative editing of its content as well as structure, while keeping a track of the corresponding changes

CHAPTER 1

INTRODUCTION

A computer needs an operating system to manage its resources and provide support for common functions such as accessing peripherals. System programming refers to writing code that takes advantage of operating system support for programmers to provide users and other programmers useful interfaces for accessing the hardware and other resources [2]. Such code comprises a class of computer software known as system software. System software provides a platform for running application software by providing a useful abstraction on top of the operating system kernel and device drivers, and is crucial to the functioning of any sufficiently complex computer system.

The writing and understanding of system software is therefore an important field of interest in computer science. As such, system programming is an integral part of any thorough computer science curriculum, including most four-year undergraduate degree programs. Yet the tools and resources for learning system programming lag behind those for learning other areas of programming, such as web development. This lack is due to the unique set of challenges involved in teaching system programming.

Section 1.1 specifies what teaching system programming involves. Section 1.2 then discusses the challenges involved in teaching system programming, the tools currently used, and why there is a need for better tools. Building upon this context, Section 1.3 concludes the chapter by introducing the focus of this thesis, a browser-based tool which aims to provide a simple, accessible, and convenient environment for learning system programming.

1.0.1 A Note About This Thesis and the Project

Software projects, by their very nature, tend to change through time. This is true especially in the front-end web development landscape, where tools and frameworks have very short lifespans.

This thesis is based on a project that is under active development, and as a result, many details such as functionality, implementation, design decisions, and development practices mentioned about the project are likely to become outdated. There is also a chance that the project may not be available in its current form in the future, although it is unlikely.

Because of this, the most important components of the project have been archived and preserved, and can be accessed through their permanent digital object identifiers (DOIs) [3],[4]. Although it is not feasible to archive every component required to be able to reproduce the developed tool exactly, the archived source code should be sufficient for understanding several details unique to the project.

Despite the fact that it is actively being developed and improved, the project in its current state is very much complete in its own right. Thus, the facts and ideas mentioned in this thesis should contribute valuable knowledge to the field of computer science.

1.1 System Programming Instruction

It is important to note that system programming is a vast topic, with varying definitions of what exactly it covers. This means that there are major differences in various system programming courses in terms of the topics covered, the tools and methods used for teaching, the placement of the course in the overall course sequence of a computer science curriculum, the expected skill-level of students, among others. Therefore, this thesis refers to system programming as described and taught in CS 241, the undergraduate system programming course offered by the Department of Computer Science at UIUC [5].

1.1.1 Platform and Programming Language

The pairing of C and Linux/Unix is used heavily by software that must provide high performance and low-level control of the program's execution [2]. This makes the combination highly suitable for introducing system programming concepts, which is why many courses, including CS 241, choose the C language running on a Linux/Unix operating system (which implements the POSIX standard) for instruction. Hence, throughout the rest of the thesis, the C over Linux/Unix combination is assumed to be the default language and platform combination for teaching system programming.

1.2 Challenges in Teaching System Programming

1.2.1 Development Environments

Because of the low-level, "close-to-hardware" nature of system programming, it requires very specific development environments, which include the operating system, compiler toolchain, the programming language, and its libraries. For example, CS 241 uses a POSIX-compliant Linux-based operating system, the clang C compiler, and other tools for debugging. The versions of each are important. This makes setting up a development environment a complicated and sensitive process, which can be daunting for beginners.

To achieve consistent development environments, many university courses ask students to use computers available in their lab facilities, or to use compatible hardware and software if students are using their own machines. One way to make setup easy and consistent (without using computer lab facilities) is by distributing pre-configured virtual machines. During several semesters CS 241 followed this approach. However, virtual machines tend to be big in size ($>$ 2 GBs), which makes them impractical to distribute to a large class or to remote students with limited Internet connection bandwidth. Another way is to provide students remote access to hosted virtual machines, via, for example, SSH. Recent offerings of CS 241 follow this approach. However, this approach is also impractical to scale to a large number of students.

The recent popularity of online courses (especially MOOCs) makes it even more important to have development environments that can be quickly bootstrapped and easily distributed. The only scalable solution is to deliver the development environment in the web browser. There are a few in-browser C programming environments [6],[7], but they usually rely on a server infrastructure to compile and execute C code, which makes them hard to scale and prone to security and privacy issues. Due to security and scalability concerns, no existing in-browser C programming environment provides full-blown access to a virtual machine, making them of little use for the purpose of system programming.

1.2.2 Course Material

System programming concepts are hard to grasp already. Getting familiar with the programming language and tools adds to the complexity for a learner, especially in an instructor-led or non-self-paced course. There is often a wide gap between concepts taught in the classroom and the skills required for hands-on practical assignments.

For example, CS 241 is the first time most computer science students at UIUC are introduced to C. However, the first assignment in the course gets started with relatively advanced concepts — even though there are lab sections where course staff can help, students are expected to learn C and navigate “trivial” programming issues by themselves.

This is where a simple-to-use C programming learning tool with short bite-sized challenges would be of great utility. There are many browser-based programming learning environments, but most of them focus on teaching web and scripting languages [8]–[11].

1.3 The Project and Its Advantages

To address the challenges mentioned in Section 1.2, we started a project that aims to build a system programming and C learning environment for beginners, which is simple to use, easy to distribute, accessible to a wide range of audiences, and easy to scale. To the best of our knowledge, prior to this project, there was no viable implementation that achieved these goals. The tool developed in the project so far has four major features and characteristics, listed in the subsections below, that are designed to achieve these goals. Recent advances and improvements in web platforms, technologies and their performance have made such a tool not only possible, but feasible as well.

1.3.1 Purely Client-Side

Being a web application, the tool does not require anything more than a recent, standards-compliant web browser. There is also no need to sign-up, login or create an account. Our tool is unique in the sense that it is a purely client-side web application — it runs entirely in the user's web browser — and so there is no need to use and maintain expensive server-side infrastructure. This means that the tool can be served by static web servers or CDNs, which are highly efficient and are often free or extremely cheap. Because of this, the application is highly scalable and easy to make available, making it well-suited for MOOCs.

1.3.2 Embedded Virtual Machine

The tool embeds a tiny but fast virtual machine, built in JavaScript, that runs a full-fledged Linux operating system right in the user's web browser. The VM contains many features and devices, such as audio, video, a proper filesystem, an emulated network interface, and more. This makes it very suitable for letting students experiment with low-level system programming, without worrying about making the system unusable. If a student makes a mistake, reverting the machine to a usable state is as simple as refreshing the web page.

1.3.3 Powerful Code Editor

The tool comes with a fully-featured code editor, with features such as syntax highlighting, multiple themes, and keyboard shortcuts. The editor has been designed to support learners new to the C language, by providing features such as automatically indenting code, quick search and access to the Linux man pages, and highlighting and linking C constructs to their corresponding man pages.

1.3.4 Integrated Learning Material

The tool also comes with lecture videos and lessons from an instructor, accompanied by short, bite-sized programming exercises to help beginners become comfortable with the C language. The current course material consists of introductory C and system programming concepts, with more content and material being continuously developed. The goal is to have a complete introductory system programming course built into the tool and available for free.

The variety and depth of features included make this tool unique and a compelling alternative to other methods of introducing system programming to beginners. The tool has immense potential and there are lots of exciting applications of this tool for students and educators as well as researchers.

CHAPTER 2

FUNCTIONALITY AND USAGE

This chapter provides a thorough tour of the tool's user interface, while demonstrating its functionality and various features. The tool can mainly be used in two ways, either as a self-paced, guided sequence of lessons and exercises, or as a free-form playground to learn and experiment with C code.

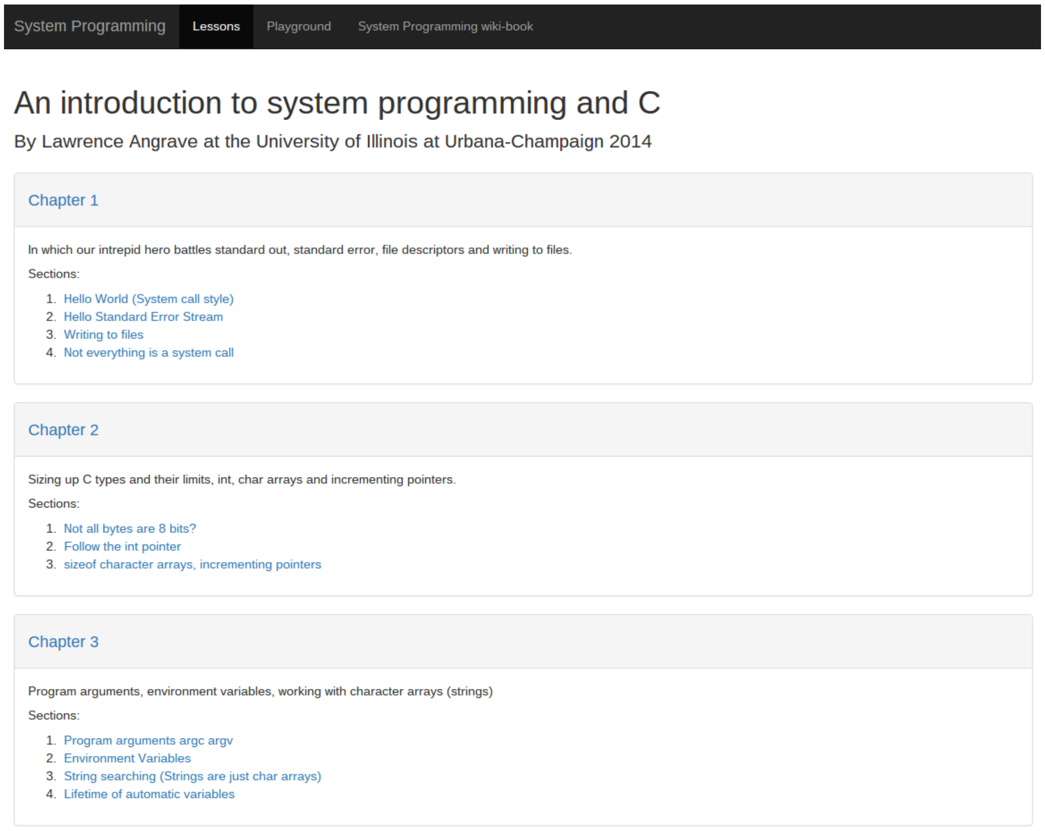
The lesson sequence is intended to be a course providing comprehensive introduction to C and system programming, and is aimed at beginners new to the C language. The course material is still being developed, but once completed, can either be used as a standalone self-paced course, or as accompanying material for a classroom course such as CS 241. Each lesson is a chapter containing multiple sections related to a topic. Each section of a given chapter consists of a series of activities. An activity could be a short lecture video, an interactive exercise, or a graded programming quiz. There are no quizzes as of yet, and interactive exercises are currently not checked to see if the student has performed them correctly. Consequently, tracking of course progress has not yet been implemented.

A resource which complements this project well is Professor Lawrence Angrave's System Programming wiki-book [12]. The wiki is a textbook-style introduction to System Programming and C. It is built by students and faculty from the University of Illinois and is a crowd-sourced authoring experiment by Professor Angrave. Being a wiki, the book is constantly being improved and expanded. CS 241 relies heavily on the wiki-book.

The web application built in this project consists of three main views or screens. A navigation bar is accessible from the top of every page, and allows navigating to any view. The navigation bar also provides a link to the aforementioned wiki-book.

2.1 Lessons Page

The lessons view lists all the available chapters, their descriptions and their sections. This is also the home page of the application, and is shown when a user first visits the website using the default URL. Figure 2.1 shows the lessons page.



An introduction to system programming and C

By Lawrence Angrave at the University of Illinois at Urbana-Champaign 2014

Chapter 1

In which our intrepid hero battles standard out, standard error, file descriptors and writing to files.

Sections:

1. Hello World (System call style)
2. Hello Standard Error Stream
3. Writing to files
4. Not everything is a system call

Chapter 2

Sizing up C types and their limits, int, char arrays and incrementing pointers.

Sections:

1. Not all bytes are 8 bits?
2. Follow the int pointer
3. sizeof character arrays, incrementing pointers

Chapter 3

Program arguments, environment variables, working with character arrays (strings)

Sections:

1. Program arguments argc argv
2. Environment Variables
3. String searching (Strings are just char arrays)
4. Lifetime of automatic variables

Figure 2.1: The Home/Lessons page

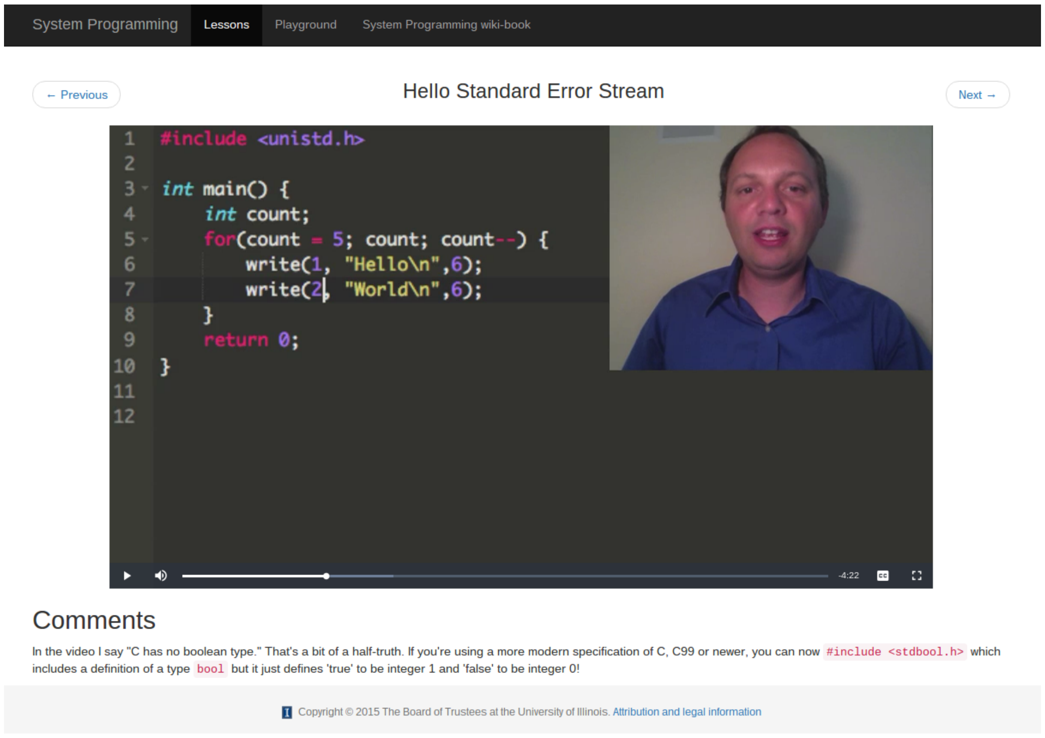
Clicking on a section title takes the user to the relevant section's first activity, usually a lecture video. Clicking on a chapter number takes the user to the first activity of that chapter's first section.

2.2 Lecture Video Page

Upon navigating to a video activity, the user is shown the video view. Figure 2.2 shows the lecture video for Chapter 1, Section 2: "Hello Standard Error

Stream". This video is the first (and only) activity of the section.

The figure showcases a number of features of the video view. The title of the video is shown above the video player. Every lecture video is followed by an optional description of the video, and optional comments made by the video's author. In this case, the lecture is given by Professor Lawrence Angrave, and is titled "Hello Standard Error Stream". Professor Angrave's comment below the video clarifies one of the concepts discussed in that video.



```
#include <unistd.h>

int main() {
    int count;
    for(count = 5; count; count--) {
        write(1, "Hello\n",6);
        write(2, "World\n",6);
    }
    return 0;
}
```

Comments

In the video I say "C has no boolean type." That's a bit of a half-truth. If you're using a more modern specification of C, C99 or newer, you can now `#include <stdbool.h>` which includes a definition of a type `bool` but it just defines 'true' to be integer 1 and 'false' to be integer 0!

Copyright © 2015 The Board of Trustees at the University of Illinois. Attribution and legal information

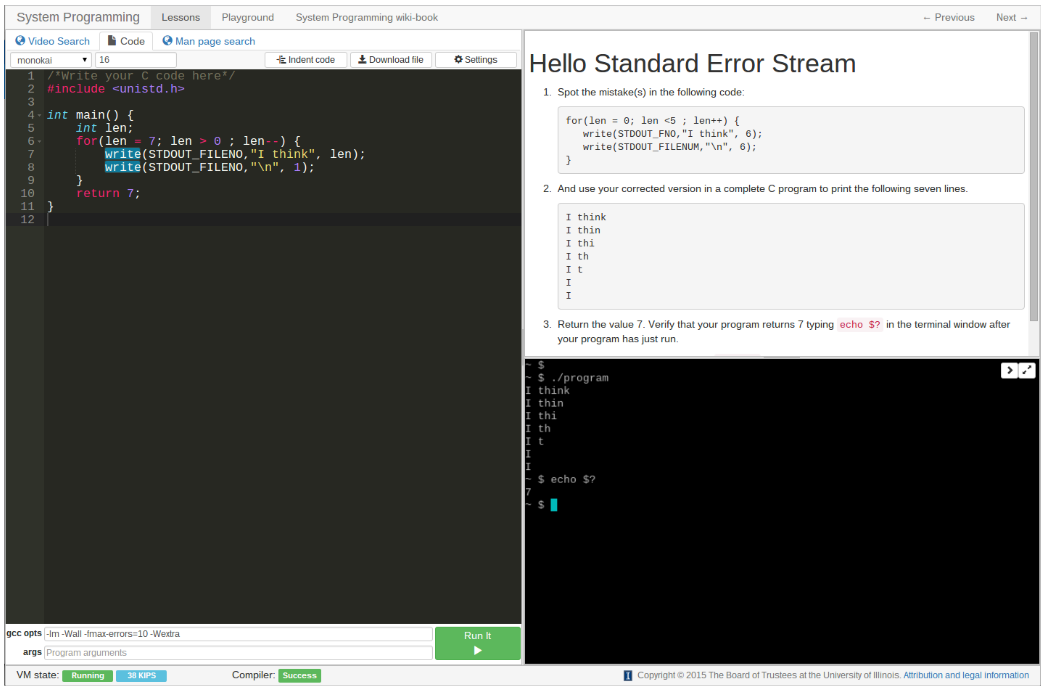
Figure 2.2: An example lecture video

The Previous and Next buttons allow the user to navigate to the previous and next activities, respectively. If the current activity is the last activity of a section, pressing Next on that page will take the user to the first activity of the next section. If the current activity is the last activity of a given chapter's last section, pressing Next will take the user to the Lessons index page. Similarly, if the current activity is the first activity of a section, pressing Previous on that page will take the user to the last activity of the previous section. If the current activity is the first activity of a given chapter's first section, pressing Previous will take the user to the Lessons index page.

Currently all lecture videos are recorded by Professor Lawrence Angrave, the current CS 241 course instructor. The video shown in Figure 2.2 is characteristic of all the other videos, in the sense that it consists of a view of Professor Angrave typing code in the playground editor, with him talking in an inset window. All currently available lecture videos are recorded in English. English subtitles are also available for each video, and can be enabled through the video player controls.

2.3 Play Activity Page

When a user navigates to a programming exercise or quiz activity (both are types of "play" activities), they are shown the play view. Figure 2.3 shows the play activity for Chapter 1, Section 2: "Hello Standard Error Stream". This activity is a programming exercise, and is the second (also the last) activity of the section.



```c
/*Write your C code here*/
#include <unistd.h>

int main() {
    int len;
    for(len = 7; len > 0 ; len--) {
        write(STDOUT_FILENO,"I think", len);
        write(STDOUT_FILENO,"\n", 1);
    }
    return 7;
}
```

gcc opts: -lm -Wall -fmax-errors=10 -Wextra

args: Program arguments

Hello Standard Error Stream

1. Spot the mistake(s) in the following code:

```c
for(len = 0; len <5 ; len++) {
   write(STDOUT_FNO,"I think", 6);
   write(STDOUT_FILENUM,"\n", 6);
}
```

2. And use your corrected version in a complete C program to print the following seven lines.

```
I think
I thin
I thi
I th
I t
I
I
```

3. Return the value 7. Verify that your program returns 7 typing `echo $?` in the terminal window after your program has just run.

```
~ $
~ $ ./program
I think
I thin
I thi
I th
I t
I
I
~ $ echo $?
7
~ $
```

Copyright © 2015 The Board of Trustees at the University of Illinois. Attribution and legal information

Figure 2.3: An example programming exercise

The Previous and Next buttons on the top-right corner of the page are identical in function to the ones in the video view, which are described in Section 2.2.

Apart from the Previous and Next buttons, the play view is identical to the playground page, and so the various elements present on the page will be described in much more detail in Section 2.4.

During an exercise, a user is expected to follow the instructions in the document in the upper-right pane, and write code in the editor to implement the program required. The user can then press the green "Run It" button to have their written code compiled and executed in the VM, as can be seen in the terminal window in the lower-right pane. Once satisfied with the results, the user can then proceed to the next activity using the Next button described earlier. As mentioned earlier in this chapter, there are currently no programming quizzes, and programming exercises are not checked to see if the student has performed them correctly. Having the ability to automatically check and grade student code is an active area of research and development in this project.

2.4 Playground

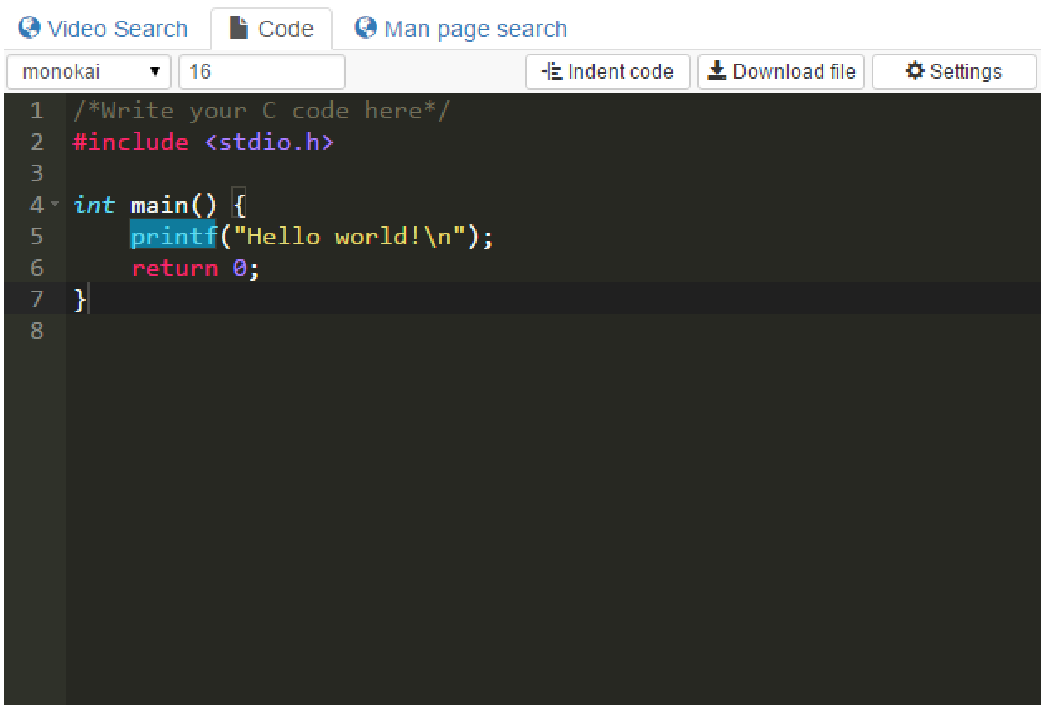
A user can reach the Playground page either by navigating directly (using the "Playground" link in the navigation bar) or by navigating to a programming exercise or quiz activity, as mentioned in Section 2.3.

The Playground is the heart of the project. This is where a user can write, compile and execute C code, and access useful documentation. Figure 2.3 in Section 2.3 shows the Playground. It has been carefully designed with the beginner C programmer in mind.

A number of details about the page are demonstrated by the figure, most notably the presence of three different panes or panels. Each pane contains various components supporting the Playground functionality. The left pane contains a tabbed browser, which contains the code editor, video search, and man page tabs. The upper-right pane contains a document explaining the current exercise. The lower-right pane contains the embedded VM's terminal window. The following subsections detail some of these components.

2.4.1 Code Editor

The playground embeds a plain-text editor, with color syntax-highlighting for C constructs, automatic indentation of code blocks, multiple themes, and more. A screenshot of the editor can be seen in Figure 2.4.

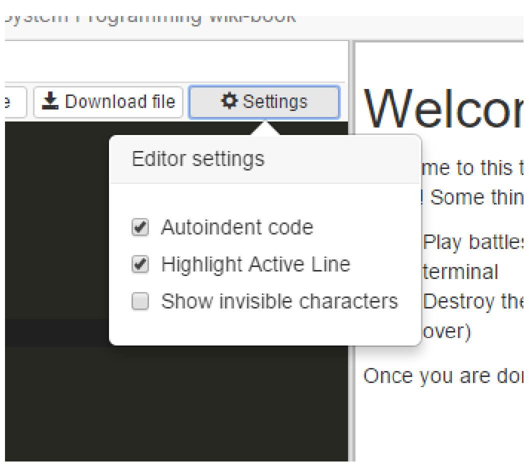


```
/*Write your C code here*/
#include <stdio.h>

int main() {
    printf("Hello world!\n");
    return 0;
}

```

Figure 2.4: The code editor

Clicking the Settings button reveals a small box for tweaking some editor preferences, as shown in Figure 2.5. These editor preferences, along with the editor theme and font size, are saved in the browser's storage, so that whenever the user visits the web application again, the editor will be exactly as they left it.

One of the most important features of the code editor is automatic highlighting of certain tokens (Linux system calls, C standard library functions and data types, etc.) which have entries in the Linux man pages. In Figure 2.4, for example, the `printf` function is highlighted. Clicking on the highlighted token opens a new "tab" containing the man page for the corresponding token. Section 2.4.2 explains more about man page tabs. To the best of our knowledge, no other C editor, web-based or otherwise, provides such direct linking to the man pages, which makes this project a unique


Editor settings

- [✓] Autoindent code
- [✓] Highlight Active Line
- [ ] Show invisible characters

Figure 2.5: The editor settings dialog

resource for beginning C programmers.

2.4.2 Man Page Search

Quick and easy access to documentation is necessary for programming, especially for beginners. Apart from automatic highlighting and linking of tokens inside the editor to corresponding man pages as mentioned in Section 2.4.1, the ability to quickly search for man pages is available from right inside the Playground, as shown in Figure 2.6.

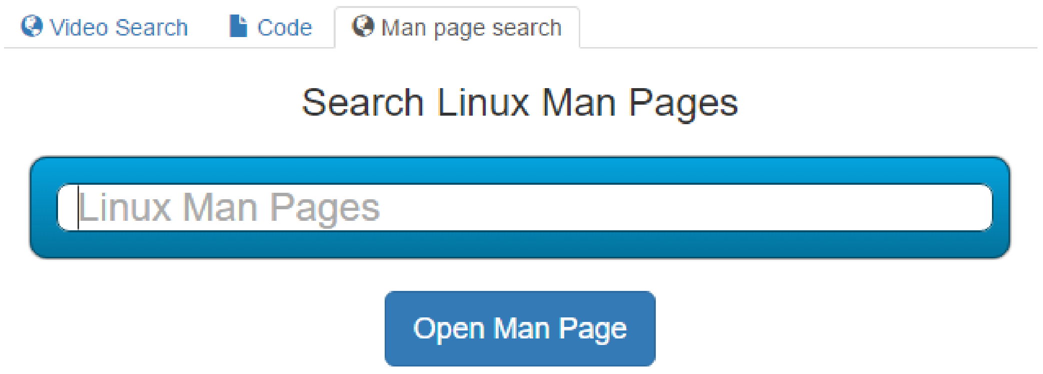
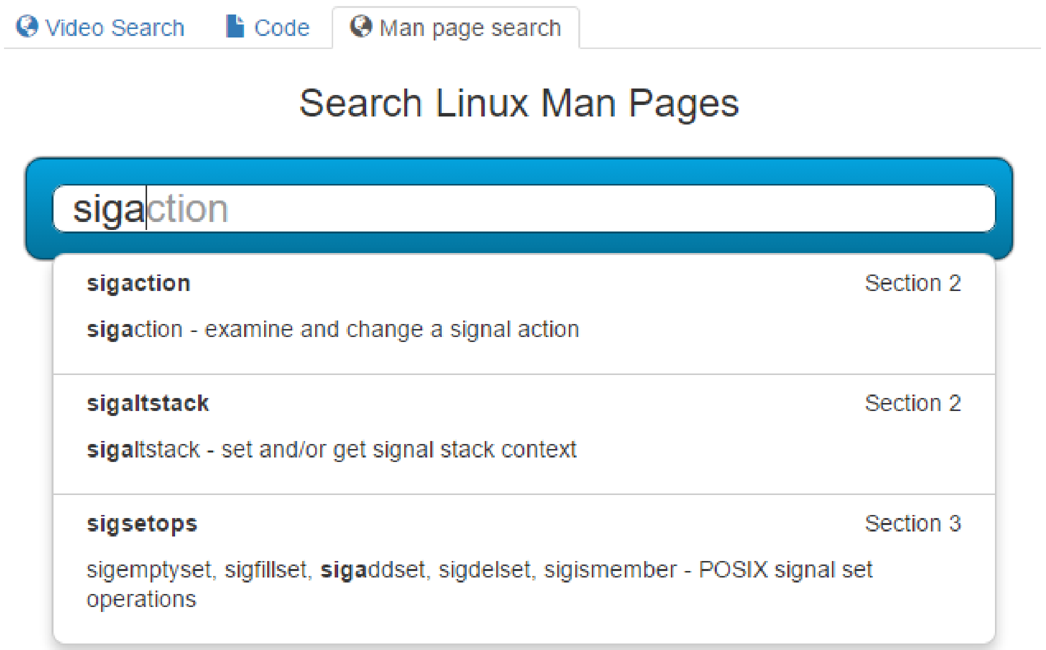


Figure 2.6: The man page search tab

To facilitate easy searching and finding the right documentation, the user's search term is automatically completed and relevant results are shown as they type. The section to which a man page belongs is also shown next to the search result. An example search is shown in Figure 2.7.

After selecting a man page, a user can either press the Enter key or click on the “Open Man Page” button to open the selected man page in a new tab. Figure 2.8 shows the man page for the `sigaction` system call open inside a tab.



sigaction — Section 2
sigaction - examine and change a signal action

sigaltstack — Section 2
sigaltstack - set and/or get signal stack context

sigsetops — Section 3
sigemptyset, sigfillset, sigaddset, sigdelset, sigismember - POSIX signal set operations

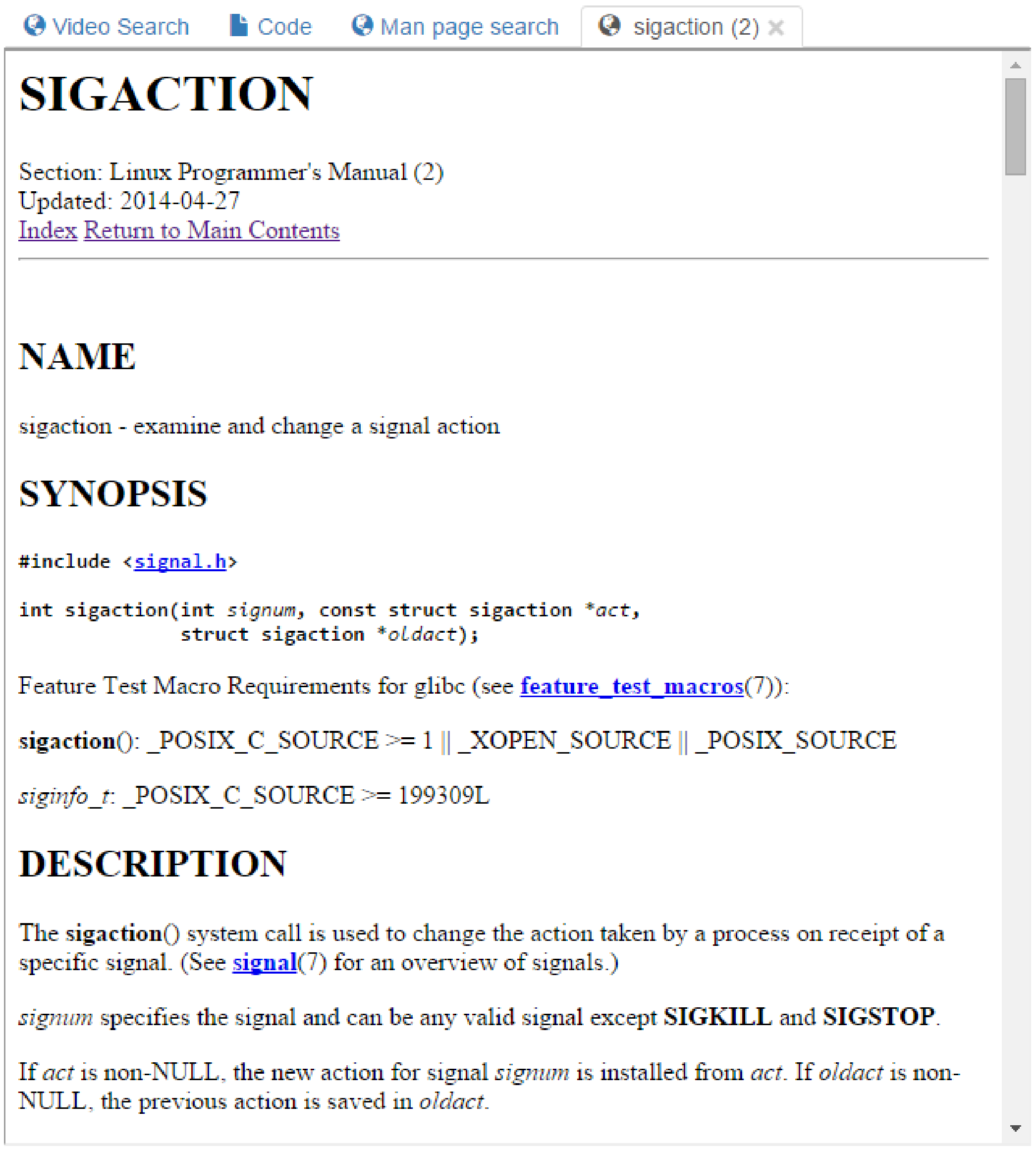
Figure 2.7: Auto-completion in action when searching for man pages

2.4.3 Virtual Machine Terminals

As mentioned in Section 1.3.2 on page 5, the project embeds a full-featured Linux-based VM. The VM has two terminals (`tty0` and `tty1`), each running a command line shell (`sh`), and can be accessed from the lower-right pane of the Playground. The terminal pane is shown in Figure 2.9.

The upper-right corner of the terminal pane contains buttons for switching the active terminal and for displaying the terminal in full-screen.

An advantage of having two terminals is that a user can run a blocking process on one terminal and still be able to interact with the VM using the



SIGACTION

Section: Linux Programmer's Manual (2)
Updated: 2014-04-27
Index Return to Main Contents

NAME

sigaction - examine and change a signal action

SYNOPSIS

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

sigaction(): _POSIX_C_SOURCE >= 1 || _XOPEN_SOURCE || _POSIX_SOURCE

siginfo_t: _POSIX_C_SOURCE >= 199309L

DESCRIPTION

The **sigaction**() system call is used to change the action taken by a process on receipt of a specific signal. (See **signal**(7) for an overview of signals.)

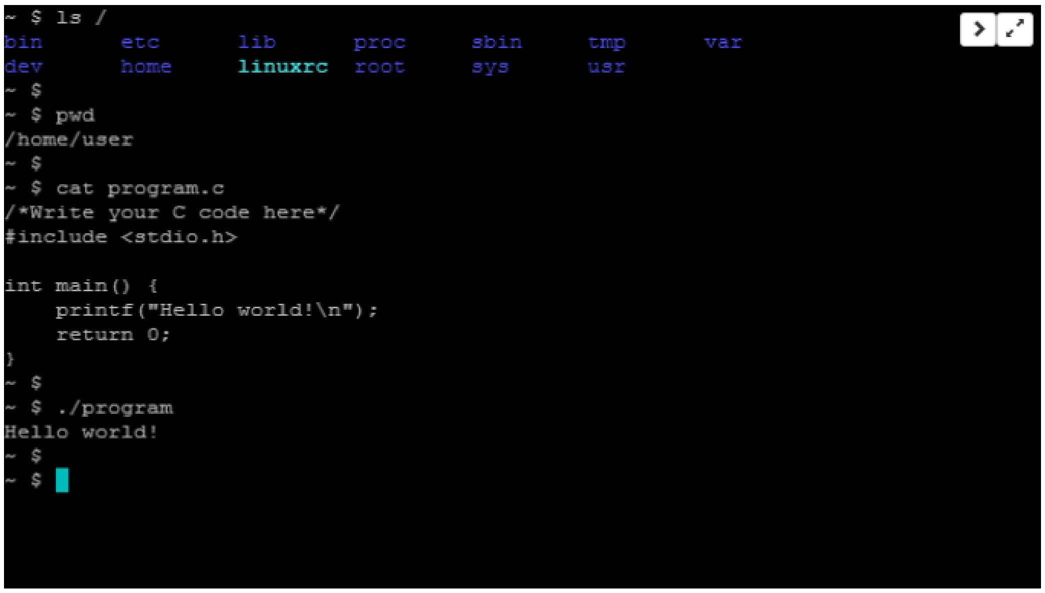
signum specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If *act* is non-NULL, the new action for signal *signum* is installed from *act*. If *oldact* is non-NULL, the previous action is saved in *oldact*.

Figure 2.8: A man page opened in a tab

other terminal. One use case of this is to have exercises where there is a server process running on one terminal and a client process running on the other terminal. These kinds of exercises might be helpful for teaching computer networking fundamentals [13],[14].

Displaying a terminal in full-screen can be useful in many cases. For example, it was used by Professor Angrave to perform live interactive command-line sessions in several CS 241 classroom lectures.

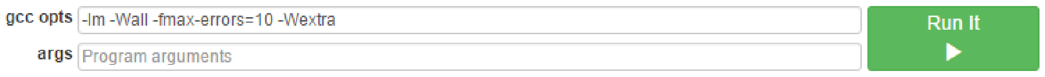


```
~ $ ls /
bin       etc       lib       proc      sbin      tmp       var
dev       home      linuxrc   root      sys       usr
~ $
~ $ pwd
/home/user
~ $
~ $ cat program.c
/*Write your C code here*/
#include <stdio.h>

int main() {
    printf("Hello world!\n");
    return 0;
}
~ $
~ $ ./program
Hello world!
~ $
~ $
```

Figure 2.9: Interacting with the VM on `tty0`

2.4.4 Compile and Run Controls

Just below the code editor is a button which allows compiling the C code present in the editor and then running it automatically. Next to the button are controls for setting options for the compiler (`gcc`), and for specifying the arguments passed to the program when running it on the command-line, as shown in Figure 2.10.



gcc opts: -lm -Wall -fmax-errors=10 -Wextra

args: Program arguments

Figure 2.10: Controls for compiling and running the program, setting options for `gcc`, and specifying the command-line arguments passed to the program

The "compile and run" command can also be invoked using the Ctrl+Enter (⌘+Enter for Mac systems) key combination inside the editor.

2.4.5 VM and Compiler Status Bar

At the bottom of the Playground screen is a status bar showing the current state of the VM, its processor speed in KIPS/MIPS if it is running, and the status of the compilation process, as shown in Figure 2.11.

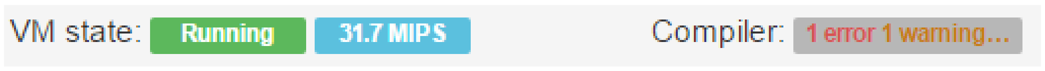


Figure 2.11: The VM and compiler status bar

Possible VM states:

Stopped The VM's execution has been explicitly stopped. The project currently does not include functionality for stopping the VM, so this status is never encountered under normal circumstances.

Booting The VM's operating system is booting.

Running The VM's operating system has finished booting and is running normally. The OS could either be running a process executed explicitly by the user, or could be waiting for user input.

Paused The VM's execution has been explicitly paused. Pausing differs from stopping the VM in the sense that the VM execution can be resumed from the exact point at which it was paused. The project currently does not include functionality for pausing the VM, so this status is never encountered under normal circumstances.

Possible compilation process states:

Waiting The VM is unavailable for compilation, because it could either be booting, or has been stopped/paused. The "compile and run" command is unavailable in this state, and the "Run It" button shown in Figure 2.10 is disabled (greyed out).

Ready The VM is running, and no compilation has been performed since the VM finished booting. The program is ready to be compiled in this state.

Compiling The program is currently being compiled. It is important to note that this status is shown only when the compilation has been started using the "compile and run" command (either through the "Run It" button or the editor keyboard shortcut). Detection of compilation status is currently not supported if the user runs the compiler directly by using the gcc command on the command-line.

Canceled The current compilation process was interrupted by invoking a new compilation. During compilation the "Run It" button is disabled, so the only way for a user to cancel the current compilation and start a new one is by using the Ctrl + Enter (⌘ + Enter for Mac systems) editor keyboard shortcut. Although the displayed status remains as "Canceled", the new compilation process will immediately be invoked and run just like in the "Compiling" status.

Success The last compilation of the program was successful. Immediately following a successful compilation, the compiled program is automatically executed on the command-line, and its output is visible in the terminal pane.

Warnings The last compilation of the program was successful, although some warnings were generated by the compiler. The status bar also displays the number of warnings generated, if it could be determined. Just like the "Success" state, the compiled program is automatically executed on the command-line following the compilation.

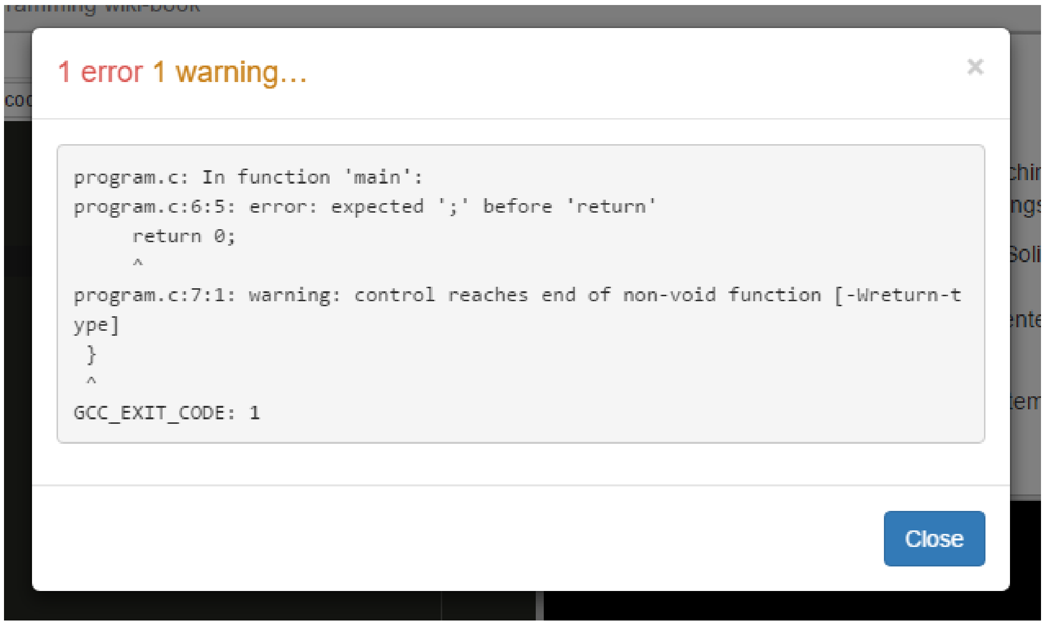
Failed The last compilation of the program was unsuccessful because of errors reported by the compiler. Depending on the nature of the errors produced (and presence of warnings), the status bar displays one of the following messages:

- "n errors...", where n is the number of errors reported: Shown when no warnings are reported
- "n errors m warnings...", where n and m are the number of errors and warnings reported, respectively
- "Failed...": Shown when errors were reported, but the number of errors could not be determined

Because the last compilation process was unsuccessful, no executable is produced by it, and consequently, no program is automatically executed after a failed compilation.

2.4.6 Compiler Error/Warning Reporting

Clicking on the compiler status box (described in Section 2.4.5) in the "Warnings" or "Failed" compile states opens a small window which shows the output of the compiler. Figure 2.12 shows an example of such a window.



```
program.c: In function 'main':
program.c:6:5: error: expected ';' before 'return'
     return 0;
     ^
program.c:7:1: warning: control reaches end of non-void function [-Wreturn-t
ype]
 }
 ^
GCC_EXIT_CODE: 1
```

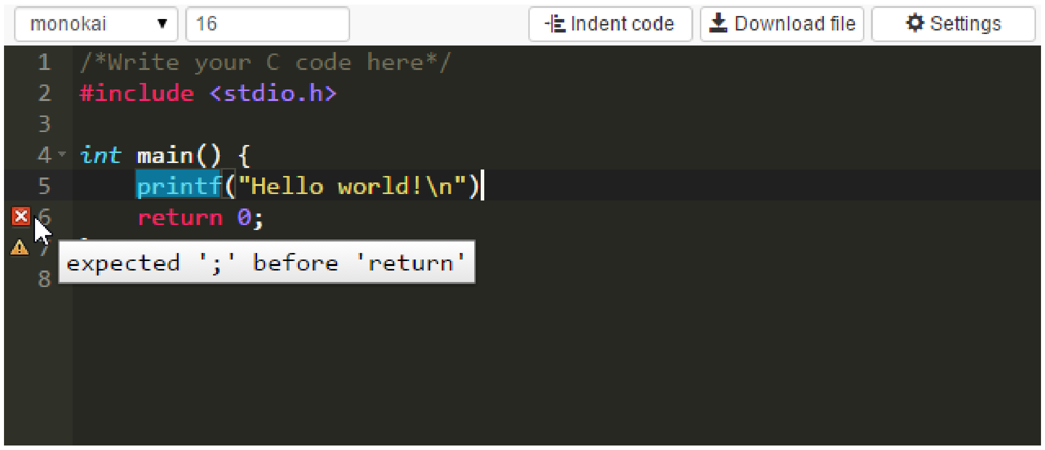
Figure 2.12: Errors/warnings reported by the compiler

If errors or warnings reported by the compiler point to lines in the program source code, those lines are annotated in the editor. The respective error/warning messages can be seen by hovering the mouse over the annotations, as can be seen in Figure 2.13.

If an error is caused by incorrect options for `gcc`, then the input box for setting compiler options is highlighted in red, and the relevant error message is shown when the box is clicked, as shown in Figure 2.14.

2.4.7 Lecture Video Search

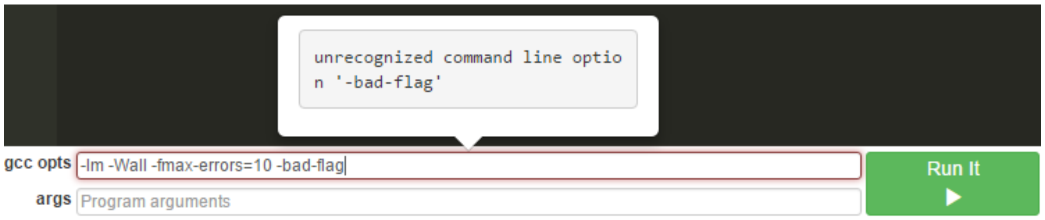
Another feature which makes this project very useful for learners is the ability to search through lecture videos based on text transcriptions of the Instructor's speech. This allows a user not only to easily locate a relevant lecture



```
/*Write your C code here*/
#include <stdio.h>

int main() {
    printf("Hello world!\n")
    return 0;
```

```
expected ';' before 'return'
```

Figure 2.13: Source code lines causing errors or warnings are annotated



```
unrecognized command line optio
n '-bad-flag'
```

gcc opts: -lm -Wall -fmax-errors=10 -bad-flag

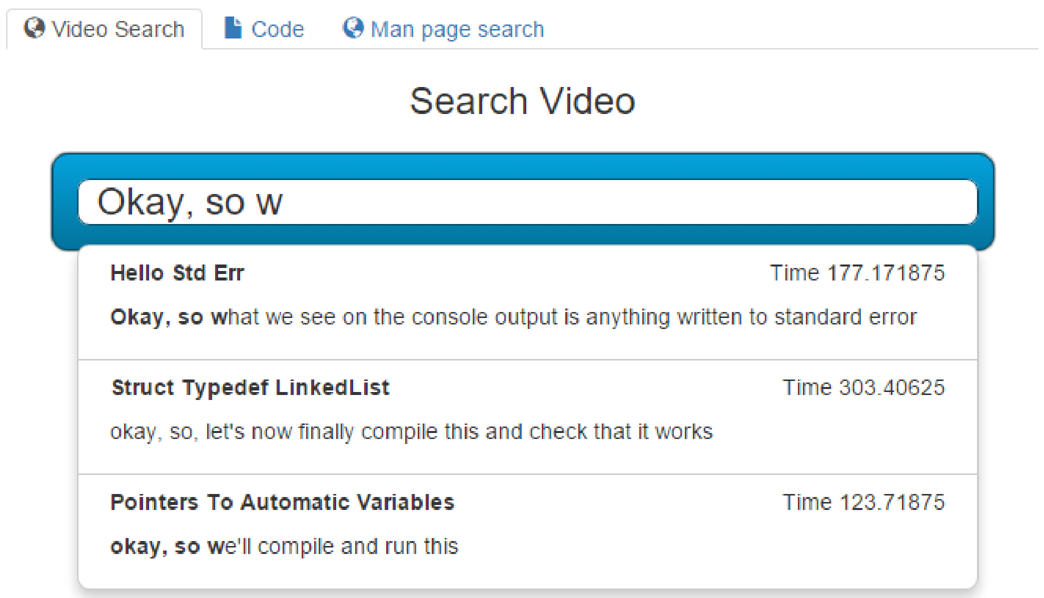
args: Program arguments

Figure 2.14: Errors in the options/flags passed to the compiler shown when the input box is clicked

video, but also jump directly to the part of the video where the instructor is saying the sentence searched for.

This functionality can be accessed through the “Video Search” tab in the Playground, as shown in Figure 2.15. As can be seen from the figure, a user can search for any part of a phrase or sentence from a lecture. Relevant search results for the search query are shown in a drop-down below the search box. Search results are filtered automatically in real-time as the search query is typed in. Every search result displays the title of the video, the phrase or sentence matching the search input, and the exact time (in milliseconds) from the start at which the matching phrase/sentence is spoken by the Instructor in the video.

After selecting the desired search result, the user can either press the Enter key or click on the “Search Video” button to see the corresponding lecture video. The resulting video is shown below the search input, as shown in Figure 2.16. The video is automatically skipped to the exact point at which the matching phrase/sentence appears.



Search Video

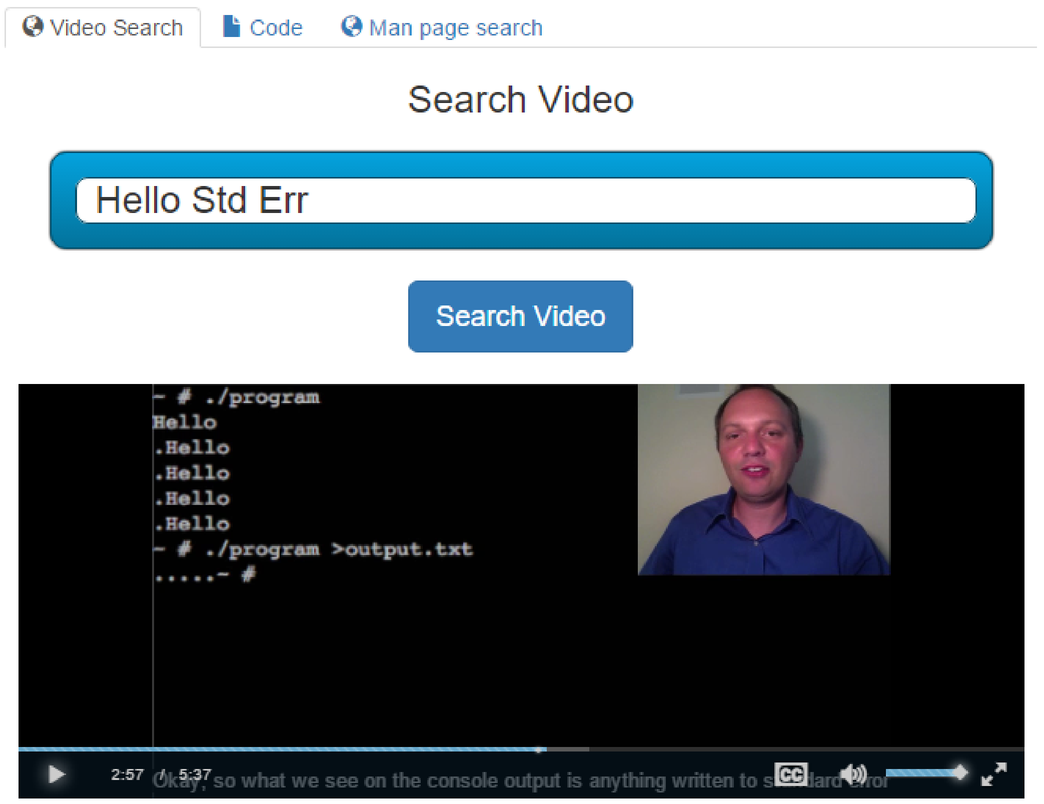
Okay, so w

Hello Std Err Time 177.171875
Okay, so what we see on the console output is anything written to standard error

Struct Typedef LinkedList Time 303.40625
okay, so, let's now finally compile this and check that it works

Pointers To Automatic Variables Time 123.71875
okay, so we'll compile and run this

Figure 2.15: Searching for a phrase in the available lecture videos



```
~ # ./program
Hello
.Hello
.Hello
.Hello
.Hello
~ # ./program >output.txt
.....~ #
```

2:57 / 5:37
Okay, so what we see on the console output is anything written to s dard error

Figure 2.16: The resulting video of a search, loaded and skipped to the relevant part

This concludes the tour of the features and functionality offered by this web application. This chapter has aimed to highlight the usefulness of this tool for learners of all skill levels.

CHAPTER 3

DESIGN AND IMPLEMENTATION

This chapter describes details of the design, architecture and implementation of the web application developed in this project. The organization of the project itself is also described, as it is important for understanding some of the technical details of the application and design decisions taken.

3.1 Design Decisions

One of the major design considerations of this project, as mentioned in Section 1.3.1, was that the entire web application should be purely client-side, that is, there should be no need to run server-side code.

This means that features traditionally requiring a back-end web server had to be implemented using workarounds to either perform the equivalent computation on the client-side (as done, for example, in the case of C code compilation), or pre-compute things and serve them statically (as done with man page metadata).

Even though a back-end server will eventually be needed for more features in the future (for example, persisting data such as user logins, user course progress, etc.), this constraint of developing a purely client-side application led to many outside-the-box solutions. These solutions make this project unique (such as having a complete VM inside the browser) and scalable (a static website is, by default, more scalable than a website needing server-side computation), and helped avoid the hassles of managing a complex web server infrastructure, as discussed in Section 3.3.

3.2 Organization of the Project

The project's source code, course material and other resources are available in public repositories hosted on GitHub, a web-based project hosting service [15]. The project also uses GitHub for collaboration, issue tracking, feature planning, code reviews, and more. The source code is managed using the Git version control system [16], which GitHub supports natively.

The project consists of five Git repositories, each hosted on GitHub under the "cs-education" GitHub Organization [17]. This Organization was created to contain these five repositories and any other repositories created by the project in the future. Today, the Organization houses several more projects developed by students and researchers at UIUC. Every project in the Organization has the common goal of improving the teaching and learning of programming and other areas of computer science.

The repositories that are part of this project (and the "cs-education" Organization) are:

- SYSBUILD (archive [3] and source [18])
- SYSASSETS [19]
- SYS-STAGING [20]
- SYS [21]
- JOR1K (archive [4] and source [22])

3.2.1 Web Application Code Base

The source code for the web application is contained in SYSBUILD, and is available under a modified version of the University of Illinois/NCSA Open Source License.[1] Issue tracking, bug reporting and feature planning for the project are also performed in this repository. The directory structure of the repository is given in Listing 3.1.

[1] `https://opensource.org/licenses/NCSA`

```
sysbuild/
├── app ........................................ Application source code
│   ├── images ....................................... Images/pictures
│   ├── jor1k .................... Jor1k dependency copied during setup
│   ├── scripts ..................................... JavaScript files
│   └── styles ................................... Sass and CSS files
├── dist ................................................ Build output
├── bower_components ................ Runtime dependencies (libraries)
├── node_modules ........... Development dependencies (Grunt plugins)
├── sys-gh-pages-config .......... Config for the deployed application
└── test ................................................. Unit tests
    └── spec
```

Listing 3.1: Directory structure of the SYSBUILD repository

The initial layout for the source code was scaffolded using a tool called Yeoman [23]. Yeoman uses "generators" to set up a project with directories, boilerplate code, automated build systems, and more, while taking into account best-practices advocated by industry leaders.

In particular, the "webapp" generator from the Yeoman team [24] was used to scaffold the project. This generator helped set up an empty project with the following components/features, all configured and wired-up to work properly with each other:

- The Bootstrap UI framework [25]
- Automated build system using the Grunt task runner [26]
- Unit testing tools and libraries
- Dependency management using Bower [27]
- CSS preprocessing using Sass [28]

3.2.2 Assets and Resources

Pre-compiled assets and resources used in the project, such as lecture videos, lesson documents, Man pages, video captions, files included in the VM filesystem, and more, are kept in SYSASSETS.

Keeping the assets separate from the main project source code allows Professor Angrave to record videos and upload them independently of the project's main repository, SYSBUILD. Several Git operations (such as cloning, pushing changes, etc) on SYSBUILD can also be performed much faster as the repository contains mostly source code, and is therefore much smaller compared to SYSASSETS. This makes the development experience for contributors much more pleasant, because most of the development work happens in SYSBUILD.

This repository is also used for serving the contained assets directly to the deployed web application, as explained in Section 3.3.3.

3.2.3 Deployment Repositories

The SYS-STAGING and SYS repositories are used for the "staging" and "production" deployment environments of the web application, respectively. Section 3.3 provides more details of the deployment setup.

3.2.4 Emulator Code Base

The JOR1K repository contains the emulator which powers the VM embedded in the application. Section 3.4.5 provides more details about the emulator and the VM.

The JOR1K repository is actually a fork of the S-MACKE/JOR1K GitHub repository [29], which is the original source of the emulator. The parent of a forked repository is usually called the "upstream" repository, which, in the case of JOR1K, is S-MACKE/JOR1K. Since the emulator is an essential part of the project, having our own fork of the upstream ensures that the project is not adversely affected in case the upstream repository is removed or its development significantly changes directions. Having a fork also allows contribution of changes back to upstream. Several contributions and changes have been made to the upstream repository as part of this project.

The work-flow practiced during development of this project is that two separate branches are maintained in JOR1K (the forked repository) — `master` and `sysbuild-stable`. The `master` branch is always kept in sync with the upstream repository's `master`. The actual dependency used by the project is kept in the `sysbuild-stable` branch. The `sysbuild-stable` branch usually lags behind `master`, and is brought up-to-date only when there is sufficient confidence that the latest changes in `master` do not break the project's functionality.

3.3 Deployment Architecture

As mentioned in Section 3.1, the entire web application is client-side, which makes it ideal for hosting on GitHub Pages, described below.

3.3.1 GitHub Pages

GitHub Pages is a free service that provides static web hosting for GitHub projects [30]. To create a project website, a branch called `gh-pages` has to be created inside the Git repository. GitHub Pages automatically makes the contents of this branch available at the URL `http://<username>.github.io/<repository>` [30]. The are several advantages of using GitHub Pages:

- Websites are highly scalable, because they are backed by GitHub's infrastructure as well as a global CDN [31]
- Support for custom domain names, although this project does not use them at the moment
- No need to manage and administer a server
- Deployment is as simple as pushing changes to a Git repository
- Because deployed sites are Git repositories, changes made during deployment can be easily tracked
- Completely free

The biggest limitation of GitHub Pages is the lack of support for server-side computation, such as processing of web requests and accessing databases, which will make it difficult to implement certain new features in the project such as the ability to save course progress of a user across different browsers or devices.

3.3.2 Repositories for Deployment to GitHub Pages

Before deployment, a build process makes several optimizations to the source code in the development repository, SYSBUILD. Instead of pushing the built code to the `gh-pages` branch of SYSBUILD for deployment, separate Git repositories are used, so as to keep SYSBUILD small and avoid it from getting bloated.

These repositories are created specifically for deployment, and contain only the `gh-pages` branch. Currently, two Git repositories are used for this purpose, as mentioned in Section 3.2.3.

3.3.3 Multi-tier Deployment Setup

Having separate repositories for deployment allows multiple deployment environments. The environments used in this project are:

Development Run from a local web server on the developer's machine. Used for testing the application during development.

Staging Hosted from the GitHub Pages site of the SYS-STAGING repository and available at `https://cs-education.github.io/sys-staging`. Used by the development team to test out new features and changes before deploying to the production environment.

Production Hosted from the GitHub Pages site of the SYS repository and available at `https://cs-education.github.io/sys`. This is the actual user-facing web application.

Assets and resources used by the application are served from the GitHub Pages site of the SYSASSETS repository, available at the URL `https://cs-education.github.io/sysassets`. All three aforementioned deployment environments use assets served from this URL.

3.4 Application Architecture

The web application is a purely client-side, single-page application. It is developed using the standard front-end web technologies: HTML, CSS and JavaScript. The project also makes use of Sass, a CSS preprocessor which adds several extensions to CSS [28].

The application is composed of five major components. Figure 3.1 shows these components and the interactions among them. Each of these components is briefly described below. It is important to note that a given component may not necessarily map to a single JavaScript module or source file.

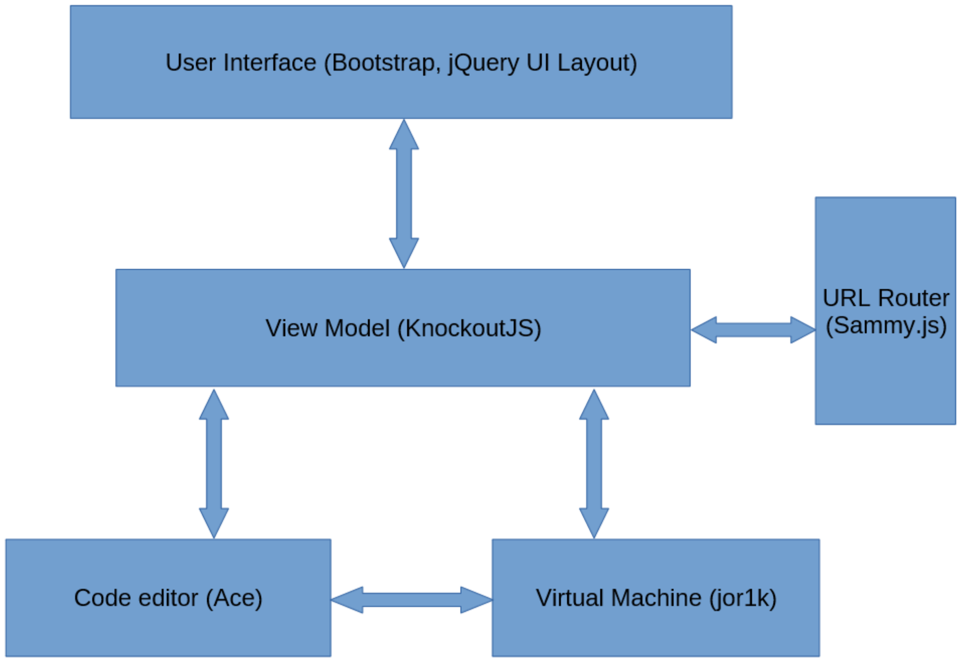


Figure 3.1: Application components and their interactions

3.4.1 User Interface

The User Interface component consists of elements displayed on the web page and the code for manipulating those elements. It also contains code for manipulating the DOM itself. It uses the Bootstrap framework [25] for various widgets and UI design elements, the jQuery library [32] for DOM manipulation and other utility functions, and the jQuery UI Layout plugin [33] for implementing the multi-pane design of the Playground (shown in Figure 2.3 on Page 10).

3.4.2 View Model

The View Model abstracts the User Interface component from the other components and also maintains certain application state. When the UI changes (for example, through user input), the View Model updates the relevant application state and notifies other components. Likewise, it updates the UI based on changes from other components. It uses the Knockout JS library [34] for two-way data-binding between a JavaScript object (the Model) and the DOM (the View).

3.4.3 URL Router

Because the web application is single-page (in fact, the entire HTML is contained in a single file), "navigation" between the various pages described in Chapter 2 is actually performed on the client-side, by showing and hiding relevant portions of the DOM and accordingly updating the web browser's URL/address bar. This kind of navigation is referred to as routing, and is handled by the URL Router component. This component uses the Sammy.js web framework [35] for performing its task, which also involves handling updates to the URL in the browser's address bar and triggering updates on the View Model component accordingly.

3.4.4 Code Editor

This component implements the C code editor described in Section 2.4.1. It is built upon Ace, an embeddable editor written in JavaScript [36]. Because the editor is a core part of this project, the underlying library powering the editor was chosen very carefully. Ace fulfilled our design requirements for many reasons, including the following:

- Ace is a mature, stable project with an active community.
- It is fully open source, under the BSD (3-Clause)[2] license [36].
- It provides good support for the C language, including syntax highlighting, auto-completion, and smart indentation of code blocks.
- It is highly extensible via an API providing thorough control over several aspects of the library.

3.4.5 Virtual Machine

This component implements the VM described in Section 1.3.2 on Page 5.

The VM is powered by jor1k, an emulator for the OpenRISC 1000 (OR1K) Instruction Set Architecture (ISA) written in JavaScript [37]. The name jor1k stands for "JavaScript OpenRISC 1000".

OR1K defines the architecture of a family of open source, RISC microprocessor cores. It is a 32/64-bit load and store RISC architecture designed with an emphasis on performance, simplicity, low power requirements, and scalability. OR1K targets medium and high performance networking and embedded computer environments [38].

Jor1k emulates the 32-bit OR1K architecture as well as several devices such as UART, audio controller, real time clock, Virtio (with support for the 9p filesystem), and more. It runs a GNU/Linux operating system based on Linux kernel 3.18 and BusyBox [39].

Like the Code editor component, the VM is a core part of this project, and so the decision to use jor1k was made after careful consideration of several alternatives. Some of jor1k's features which make it a good fit for this project are:

- It is very fast (in fact, Section 4.1 on Page 43 shows that it is much faster than alternative JavaScript VMs, which is what makes jor1k a

[2]`https://opensource.org/licenses/BSD-3-Clause`

clear choice for this project).

- It is completely open source, under the BSD (2-Clause)[3] license [29].
- It has out-of-the-box support for Linux.
- OR1K is a popular and simple architecture.
- The lead developer is very supportive and responsive.
- The project is actively developed and has a sufficiently large community.
- The code base is well-written, extensible and easy-to-understand.
- It provides support for networking (through an emulated Ethernet device using WebSockets), although our project does not use it currently.
- It provides support for graphics (through the Linux framebuffer, `fbdev`), although our project does not use it currently.

3.5 Implementation of the "Compile and Run" Flow

Details of the implementations of each of the components mentioned in Section 3.4 and their interactions are beyond the scope of this thesis. However, it is worthwhile to explain how the VM, C compiler, code editor, and other parts of the Playground work together to support compiling and running the user's code.

The following explanation refers mainly to the files `app/scripts/live-edit.js` and `app/scripts/sys-runtime.js`, part of the web application's source code in the SYSBUILD repository. A few code snippets are shown to aid in the explanation. It should be noted that these snippets contain modified versions of the actual source code. Modifications include omission of parts of code not relevant to the explanation, and some formatting changes. As such, these code snippets cannot be run independently. The language used in the snippets, like the rest of the application, is JavaScript (in particular, EcmaScript 5.1.[4])

[3] https://opensource.org/licenses/BSD-2-Clause
[4] http://www.ecma-international.org/ecma-262/5.1/

3.5.1 Sending Commands to the VM

Sending commands to the VM is fairly straightforward — jor1k supports sending data directly to the terminal. However, things get complicated when there is a need to wait for the command to finish executing, and/or capture its output. Because of the asynchronous nature of JavaScript, one cannot block the "main thread" to wait for the command to finish. Also, except for inefficient polling, there is no direct way to find out if and when a command has finished executing, because jor1k does not provide any notifications for events occurring in the VM's operating system.

We implemented the `sendKeys` function to work around these limitations. Listing 3.2 shows the signature of this function.

The code shown in Listing 3.5 makes use of the "expect" string argument of `sendKeys` to determine when GCC has exited, as explained later in Section 3.5.2.

3.5.2 Invoking the Compiler

When a user clicks the "Run It" button or presses the Ctrl+Enter (⌘+Enter for Mac systems) key combination inside the editor, the `compile` function is executed, shown in Listing 3.3. This function then invokes the `runCode` method on the `liveEdit` object,[5] passing in the program's source and the options to be passed to GCC (see Section 2.4.4).

The `runCode` function, shown in Listing 3.4, validates the code input, updates the UI with the new compilation status (see Section 2.4.5), and invokes the `startGccCompile` function on the `runtime` object,[6] passing in the validated code, the compiler options, and a callback function to be invoked once the compilation has finished.

The `startGccCompile` function, shown in Listing 3.5, enables capturing of the text flowing into the terminal, by setting `this.captureOutput = true`.

[5]The `liveEdit` object is defined elsewhere in the source code. For the purposes of this discussion, it can be simply assumed to be a module containing some functions.

[6]Similar to `liveEdit`, the `runtime` object is defined elsewhere in the source code, but can be assumed to be a module containing some functions.

It then makes use of the `sendKeys` function (described in Section 3.5.1) to delete any existing `~/program.c` and `~/program` files created during previous compilations, using the `rm` command. Next, it uses the `sendTextFile` function, shown in Listing 3.6, to copy the contents of the editor to the `~/program.c` file inside the VM.

Finally, the function invokes the C compiler using the `sendKeys` function. As can be seen in Listing 3.5, the command sent to the VM contains some `echo` and `clear` commands preceding and following the actual `gcc` command. Because all the commands are sent as a single command string (separated by semicolons), the shell executes them one-by-one, and each command's output is followed by the next without the terminal prompt being inserted between the two outputs. Because terminal output capture was enabled earlier, the output of this compound command is captured and saved. An example of the output text captured is shown in Listing 3.7, where it can be seen that the output of the `gcc` command is "fenced" between the `###GCC_COMPILE...` "magic" strings. Using this fact and the "expect" parameter of `sendKeys`, the `compileCb` callback is registered to be called when compilation has finished.

3.5.3 Post-Compilation

As mentioned in Section 3.5.2 above, the `compileCb` callback function (shown in Listing 3.8) is called when compilation has finished. This function exploits the format of the captured terminal output (example shown in Listing 3.7) to extract the output of the `gcc` command as well as its exit code using Regular Expressions. Using a helper module (not listed), the function then parses this output to generate a list of errors, warnings, and other information. This information is passed as a parameter to the `processGccCompletion` callback function registered by the `runCode` function in Listing 3.4.

The `processGccCompletion` function, shown in Listing 3.9, uses the result of the compilation passed from `compileCb` to update the UI. Next, based on the exit code of the `gcc` process, the function determines whether the compilation succeeded or failed, and sets the compile status accordingly. If the compilation was successful, the function invokes the `startProgram` function with the name of the executable file ("`program`") and the argument string to be passed on the command line (see Section 2.4.4).

The `startProgram` function, shown in Listing 3.10, sanitizes its input and then sends the command to run the compiled executable. Throughout the entire compilation process, several `clear` commands are placed appropriately, so that the terminal does not become cluttered with any of the “magic” strings, etc. So, once the `startProgram` function has executed, only the output of the executable run on the command line is visible to the user in the terminal pane (see Section 2.4.3). At this point, the “Compile and Run” flow is complete.

```
/**
 * Send a command text to the specified jor1k terminal.
 * Optionally register a callback function to be called
 * when a specified string is output on the terminal,
 * useful for getting notified when the command has
 * finished executing.
 *
 *
 * @param {string} tty - The terminal ('tty0' or 'tty1')
 *     on which the text is sent
 *
 * @param {string} text - The text to be sent to tty
 *
 * @param {string} expect - The string to listen for in the
 *     terminal output
 *
 * @param {function} success - Callback function to be
 *     called when either the expect string has been found,
 *     or when .cancel() has been called on the return
 *     value of this function
 *
 * @param cancel - Unused
 *
 * @return {object} An object providing a .cancel() method
 *     to remove the listener for the expected string
 */
SysRuntime.prototype.sendKeys =
        function (tty, text, expect, success, cancel) {
    // ...
};
```

Listing 3.2: Signature of the `sendKeys` function

```
var compile = function () {
    var code = editor.getText();
    var gccOptions = viewModel.gccOptions();
    liveEdit.runCode(code, gccOptions);
};
```

Listing 3.3: The `compile` function, executed when the “Run It” button is clicked or the editor keyboard shortcut is used

```
LiveEdit.prototype.runCode = function (code, gccOptions) {
    // check for valid code input
    if (code.length === 0
        || code.indexOf('\x03') >= 0
        || code.indexOf('\x04') >= 0) {
        return;
    }

    var callback = this.processGccCompletion.bind(this);
    this.viewModel.compileStatus('Compiling');
    this.runtime.startGccCompile(code, gccOptions, callback);
};
```

Listing 3.4: The `runCode` function

```
SysRuntime.prototype.startGccCompile =
        function (code, gccOptions, guiCallback) {

    // ensure the VM has finished booting
    if (!this.bootFinished) { return 0; }
    // remove callbacks for any previous compilations
    if (this.expecting) { this.expecting.cancel(); }

    this.ttyOutput = '';
    this.captureOutput = true;
    ++this.compileTicket;

    // delete existing source file and executable,
    // if present
    this.sendKeys('tty0',
        '\x03\ncd ~;rm program.c program 2>/dev/null\n');

    this.sendTextFile('program.c', code);

    var cmd = 'echo \\#\\#\\#GCC_COMPILE\\#\\#\\#;clear;gcc '
            + gccOptions
            + ' program.c -o program; echo GCC_EXIT_CODE: $?;'
            + ' echo \\#\\#\\#GCC_COMPILE_FINISHED\\#\\#\\#'
            + this.compileTicket + '.;clear\n';

    this.expecting = this.sendKeys('tty0',
        cmd,
        'GCC_COMPILE_FINISHED###' + this.compileTicket + '.',
        compileCb);

    return this.compileTicket;
};
```

Listing 3.5: The `startGccCompile` function, which invokes the actual `gcc` command

```
SysRuntime.prototype.sendTextFile =
        function (filename, contents) {

    this.sendKeys('tty0',
        '\nstty raw\ndd ibs=1 of=' + filename
        + ' count=' + contents.length + '\n' + contents
        + '\nstty -raw\n');
};
```

Listing 3.6: The `sendTextFile` function, used to write text data to a file inside the VM (using the `dd` command)

```
###GCC_COMPILE###
program.c: In function 'main':
program.c:5:5: warning: implicit declaration of function
 ↪  'printf' [-Wimplicit-function-declaration]
     printf("Hello world!\n");
     ^
program.c:5:5: warning: incompatible implicit declaration of
 ↪  built-in function 'printf'
GCC_EXIT_CODE: 0
###GCC_COMPILE_FINISHED###1.
```

Listing 3.7: An example of the terminal output captured after `startGccCompile` sends the compilation command (some text has been stripped from the beginning to show the relevant part)

```
this.gccOutputCaptureRe =
/###GCC_COMPILE###\s*([\S\s]*?)\s*###GCC_COMPILE_FINISHED###/;

this.gccExitCodeCaptureRe = /GCC_EXIT_CODE: (\d+)/;

// called when GCC has exited
var compileCb = function (completed) {
    var result = null;
    this.expecting = undefined;
    if (completed) {
        this.captureOutput = false;

        // get the GCC output text
        var regexMatchArray =
            this.gccOutputCaptureRe.exec(this.ttyOutput);
        var gccOutput = regexMatchArray[1];

        // get the GCC process' exit code
        var gccExitCode = parseInt(
            this.gccExitCodeCaptureRe.exec(gccOutput)[1]);
        this.ttyOutput = '';

        // ...
        // parse GCC output, then get stats
        // (error/warning counts) and editor annotations
        // (markings of lines with errors/warnings in file)
        // ...

        result = { exitCode: gccExitCode,
                   stats: stats,
                   annotations: annotations,
                   gccOutput: gccOutput };
    }
    guiCallback(result);
}.bind(this);
```

Listing 3.8: The `compileCb` function, called after the `gcc` process has exited

```
LiveEdit.prototype.processGccCompletion = function (result) {
    // result = { 'exitcode':gcc_exit_code,
    //            'stats':stats,
    //            'annotations':annotations,
    //            'gcc_ouput':gcc_output }
    // OR null if compilation was cancelled

    this.viewModel.gccErrorCount(0);
    this.viewModel.gccWarningCount(0);

    if (!result) {
        // cancelled
        this.viewModel.compileStatus('Cancelled');
        return;
    }

    // clear the terminal
    this.runtime.sendKeys('tty0', 'clear\n');

    // ... update UI with the results ...

    if (result.exitCode === 0) {
        this.viewModel.compileStatus(
            result.stats.warning > 0 ? 'Warnings':'Success');

        this.runtime.startProgram('program',
            this.viewModel.programArgs());
    } else {
        this.viewModel.compileStatus('Failed');
    }
};
```

Listing 3.9: The `processGccCompletion` function, called after `gcc` output has been parsed and error/warning information has been extracted

```
SysRuntime.prototype.startProgram =
        function (filename, cmdargs) {

    if (!filename) {
        return;
    }
    // escape strings
    if (filename[0] !== '/' && filename[0] !== '.') {
        filename = './' + filename.replace(' ', '\\ ');
    }
    cmdargs = cmdargs
                .replace('\\', '\\\\')
                .replace('\n', '\\n');

    this.sendKeys('tty0',
        '\n' + filename + ' ' + cmdargs + '\n');
};
```

Listing 3.10: The `startProgram` function, used to execute a program; invoked after a successful compilation

CHAPTER 4

RESULTS AND ANALYSIS

This chapter describes several metrics pertaining to the usage and performance of the application, with the goal of providing useful information to someone looking to use and/or deploy the application for their own purpose.

4.1 Jor1k Performance Comparison

The jor1k project has documented the results of a benchmark performed in September 2015 comparing the performances of jor1k and two other popular JavaScript-based emulators [40]. This section describes the benchmark and analyses its results.

It is important to note that the benchmark and results below are taken verbatim from jor1k's documentation, and are assumed to be accurate. No attempt was made to reproduce the benchmark results for the purpose of this thesis. As such, these results should not be considered as the output of rigorous research, but instead should only be used as anecdotal evidence. However, the results are in complete agreement with our own experience with the benchmarked systems.

4.1.1 Host System

The benchmark was performed on a system with the following characteristics:

Processor Intel® Core™2 Duo E8400 3.0 GHz

Operating System Windows 10

Web browser Mozilla Firefox 41.0

4.1.2 Benchmarked Systems

The following three systems were benchmarked:

1. jor1k running Linux 4.1 and Busybox
2. jslinux running Linux 2.6.20 and Busybox[1]
3. v86 running Linux 4.0 and GNU packages (`gzip` and `bzip2`) from Archlinux[2]

4.1.3 Benchmark Tests

The following three tests were run on each of the systems being benchmarked, and the time taken to execute each test was recorded:

1. Generate a file containing 1 MiB of random data, using the command: `dd if=/dev/urandom of=file bs=1M count=1`.
2. Compress the file generated in Test 1 using the `gzip` program.
3. Compress the file generated in Test 1 using the `bzip2` program.

Each test was executed several times on a given system, but only the lowest measured timing for that test and system combination was included in the results.

4.1.4 Results

Figure 4.1 shows the results of the benchmark. As is clear from the figure, jor1k is considerably faster than both jslinux and v86.

[1]`http://bellard.org/jslinux/`
[2]`http://copy.sh/v24/`

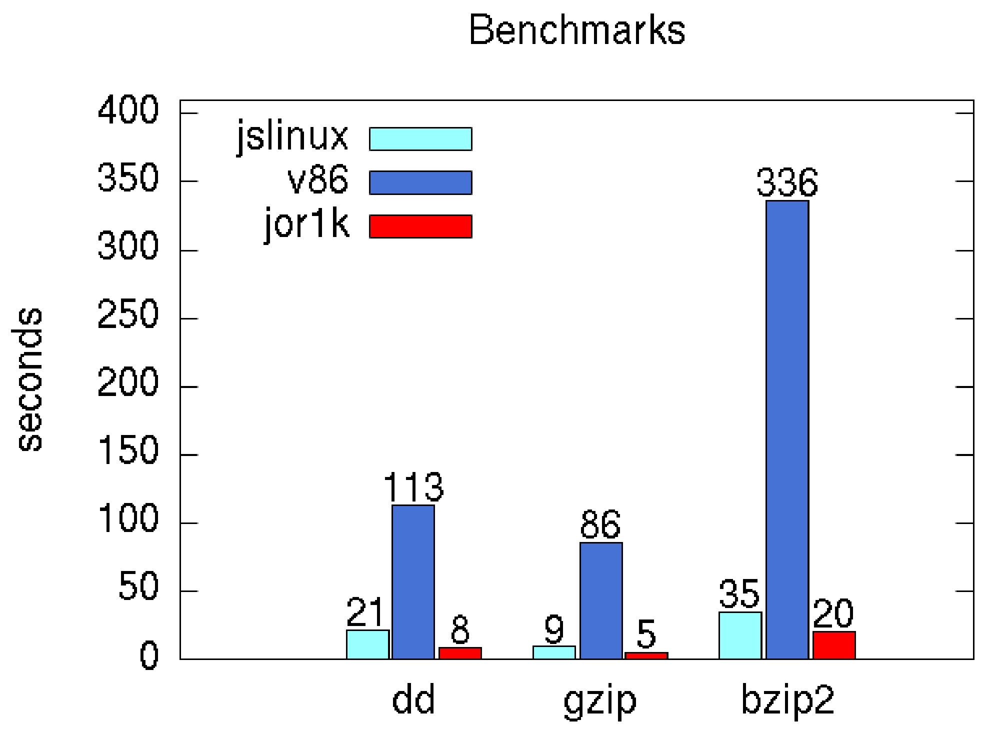


Figure 4.1: Comparison of the running times of various benchmark tests run on popular JS-based emulators (smaller values are better)

4.2 File Size Statistics

The total size ("weight") of a web application is an important factor to consider when deciding its suitability for a given use case. Applications needing to download large files can create accessibility barriers for users with limited/low bandwidth Internet connections, although the caching performed by web browsers can reduce this impact significantly.

For the web application developed in this project, Table 4.1 shows the list of files downloaded by the browser over a duration spanning from the time the user visits the application, to the time when a simple C program (shown in Listing 4.1) has been compiled and run. The type of a request or file retrieved is mentioned next to the file name, along with its size.

The table was generated from network requests captured using Chrome DevTools, a set of web authoring and debugging tools built into the Google Chrome web browser (in particular, Chrome 64-bit Version 47.0.2526.106 was used). Caching was disabled so that requests could be captured accurately. Data URLs are not included in the table, as they are always retrieved from the cache. It should be noted that the size column in the table refers to the sizes of the actual data transferred "over the wire", which contains response headers along with the content. This data may or may not be compressed, and hence the sizes shown are not the same as the sizes of the files themselves. The number of bytes downloaded on the network is shown because it is more relevant for determining the accessibility of a web application.

Jor1k loads parts of the filesystem on demand, which can be seen as XHR requests (such as downloading `clear.bz2` when the `clear` command is used). Due to this "lazy loading", it is not possible to provide an upper limit on the total size of files downloaded by the application. For example, compilation of a C program which includes a header file not already loaded will result in an additional download of the header file, along with its dependencies. However, the bulk of the total download size consists of files required for any compilation (such as the Linux kernel and the compiler), which means that the total download size should not significantly exceed the total size mentioned in Table 4.1.

Table 4.1: Network Requests Made from Page Load to Program Execution (in Chronological Order)

Name	Type	Size
`sys/`	document	4.4 KiB
`aa065ef4.vendor.css`	stylesheet	7.3 KiB
`a3077974.main.css`	stylesheet	24.4 KiB
`775af888.modernizr.js`	script	4.3 KiB
`a4093255.imark_bold.gif`	gif	1.4 KiB
`7e11eee2.vendor.js`	script	330 KiB
`09c083e1.plugins.js`	script	11.1 KiB
`a1ebcd5e.main.js`	script	20.4 KiB
`analytics.js`	script	11.0 KiB
`glyphicons-halflings-regular.woff2`	font	18.2 KiB
`jor1k-worker-min.js`	script	79.6 KiB
`sys_man_page_index.min.json`	xhr	25.9 KiB
`sys_man_page_index.min.json`	xhr	25.9 KiB

Continued on next page

Table 4.1 – continued from previous page

Name	Type	Size
`sys.min.json`	xhr	1.8 KiB
`transcription_index.min.json`	xhr	53.4 KiB
`notific8.woff`	font	29.6 KiB
Google Analytics page view	gif	373 B
`vmlinux.bin.bz2`	xhr	2.0 MiB
`basefs-compile.json`	xhr	1.1 KiB
`busybox.bz2`	xhr	612 KiB
`nsswitch.conf`	xhr	715 B
`group`	xhr	626 B
`fstab`	xhr	770 B
`inittab`	xhr	830 B
`host.conf`	xhr	668 B
`inetd.conf`	xhr	690 B
`passwd`	xhr	734 B
`interfaces`	xhr	659 B
`rcS-compile`	xhr	1.5 KiB
`services`	xhr	19.0 KiB
`profile-compile`	xhr	613 B
`default.script`	xhr	1.8 KiB
`fs.json`	xhr	40.0 KiB
`gcc.bz2`	xhr	317 KiB
`as.bz2`	xhr	130 KiB
`libmenu.so.5.9.bz2`	xhr	14.7 KiB
`libncurses.so.5.9.bz2`	xhr	149 KiB
`libbfd-2.24.51.20140817.so.bz2`	xhr	294 KiB
`libc.so.bz2`	xhr	384 KiB
`libgcc_s.so.1.bz2`	xhr	137 KiB
`stdio.h`	xhr	5.7 KiB
Google Analytics page view	gif	386 B
`clear.bz2`	xhr	2.3 KiB
`cc1.bz2`	xhr	4.2 MiB
`libz.so.1.2.8.bz2`	xhr	42.6 KiB

Continued on next page

Table 4.1 – continued from previous page

Name	Type	Size
`features.h`	xhr	1.3 KiB
`alltypes.h`	xhr	10.4 KiB
`libopcodes-2.24.51.20140817.so.bz2`	xhr	48.3 KiB
`collect2.bz2`	xhr	232 KiB
`ld.bfd.bz2`	xhr	147 KiB
`crt1.o.bz2`	xhr	1.9 KiB
`crti.o.bz2`	xhr	866 B
`crtbegin.o.bz2`	xhr	2.1 KiB
`libm.a`	xhr	602 B
`libgcc.a.bz2`	xhr	165 KiB
`crtend.o.bz2`	xhr	1.3 KiB
`crtn.o.bz2`	xhr	836 B
Total number of requests: 57	**Total size: 9.5 MiB**	

```c
#include <stdio.h>

int main() {
    printf("Hello world!\n");
    return 0;
}
```

Listing 4.1: The C program compiled while network requests were being captured

4.3 Usage and Adoption

The web application uses Google Analytics [41] for tracking various important metrics related to its usage.

Figure 4.2 graphs (in blue) the total number of sessions[3] on the Production application (`https://cs-education.github.io/sys`) per week from

[3]A session is the period of time a user is actively engaged with the website.

January 1, 2015, to November 30, 2015.

The figure also shows (in orange) the number of sessions originating from a desktop/laptop computer located in either Champaign or Urbana.[4] These sessions should be fairly representative of UIUC students visiting the application. As can be seen from the figure, these sessions represent a majority of the total sessions.

An interesting pattern to note is that the number of sessions is high at the beginning of the academic semesters at UIUC (the beginnings of the Spring 2015 and Fall 2015 semesters are marked in the figure), and very low in the period between the end of Spring semester (also marked) and the beginning of Fall semester, which corresponds to the summer break. Professor Angrave introduces the application to the new batch of CS 241 students at the beginning of each semester, which could probably have caused this pattern of usage.

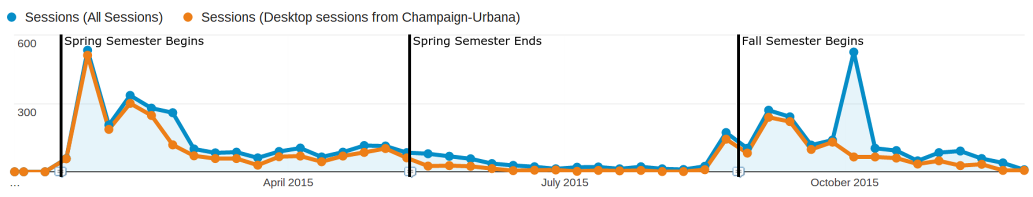
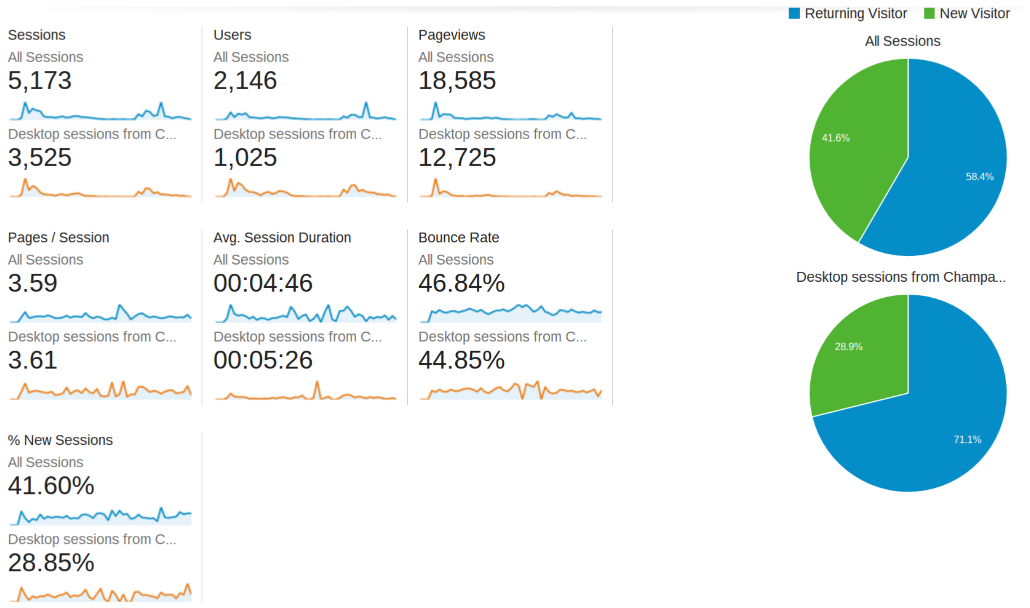


Figure 4.2: Total number of sessions (blue) on the Production website and number of desktop sessions from Champaign-Urbana (orange) from January 1, 2015, to November 30, 2015

Figure 4.3 shows a few more metrics related to user engagement on the Production website, for the same date range as Figure 4.2. Further analysis is required to be able to extract meaningful insight out of this information, which is beyond the scope of this thesis.

[4]The UIUC campus is located in the Champaign and Urbana twin-cities.


Sessions
All Sessions
5,173
Desktop sessions from C...
3,525

Users
All Sessions
2,146
Desktop sessions from C...
1,025

Pageviews
All Sessions
18,585
Desktop sessions from C...
12,725

Pages / Session
All Sessions
3.59
Desktop sessions from C...
3.61

Avg. Session Duration
All Sessions
00:04:46
Desktop sessions from C...
00:05:26

Bounce Rate
All Sessions
46.84%
Desktop sessions from C...
44.85%

% New Sessions
All Sessions
41.60%
Desktop sessions from C...
28.85%

🟦 Returning Visitor 🟩 New Visitor
All Sessions
41.6%
58.4%
Desktop sessions from Champa...
28.9%
71.1%

Figure 4.3: Several key Google Analytics metrics for the Production website from January 1, 2015, to November 30, 2015

CHAPTER 5

IMPROVEMENTS AND SUGGESTIONS

This chapter provides a few suggestions for making the project more useful as a learning tool.

5.1 Automatic Grading of Student Code

The tool will become more useful if it can provide some feedback on code written by a student. One way of providing feedback is to automatically evaluate, analyze, and grade the code. Compared to manual review by an instructor, this provides instant feedback and is more scalable and practical. There are quite a few possible options for performing such automated assessment.

A simple and effective way is to check the compiled program's output, for a given set of inputs, to see if it matches the expected output. The output of a program consists of text printed on the screen (`stdout` or `stderr`), data written to files on the disk, data sent through a socket, etc. Likewise, the input of a program consists of arguments passed through the command line, text entered on `stdin`, data read from files on the disk, data received through a socket, etc.

Deeper insight into the student's code can be gained by tracking whether the right library or system calls have been made, and with the right parameters. This can be done using the `ltrace` or `strace` Linux utilities.

Tischer [42] lists more ways of automatically assessing student code, including static analysis and symbolic execution, and discusses some approaches for improving the current methods.

5.2 Event Tracking and Analytics

Currently, the number of page views is tracked and analyzed, as detailed in Section 4.3. However, it is difficult to see how students are actually using the application, and whether they are benefiting from it, based on page views alone. A lot of insight can be gained by tracking and analyzing a user's behavior and actions taken inside the web application. The application's ease of deployment and its potentially wide reach open up many exciting research opportunities for answering questions such as how and where students are using this tool, the effectiveness of this tool compared to traditional classrooms, and more.

The following sub-sections list some interesting events to track, and the potential analytical insights they can provide.

5.2.1 Video Views

Here are some example events that can be tracked when a user watches a lecture video: "video played", "25% video viewed", "50% video viewed", "75% video viewed", and "100% video viewed".

These events can provide information such as the percentage of users who watch a given lecture video to completion, the average length of a given lecture video seen before users move on to the corresponding exercise, etc. This information can be used to evaluate the quality of lectures, identify topics which need more explanation, and more.

5.2.2 Number of Code Compilations

Some sample metrics to keep track of: Number of times a user initiates a compilation, amount of time each compilation takes, number of successful compilations, number of compilations with warnings, and so on.

Based on these metrics, one could find, for example, the average number of mistakes made by a user when writing a program.

CHAPTER 6

CONCLUSION

The project described in this thesis started out as an experiment to see if it was possible to compile and run C programs completely in the browser, and whether speeds acceptable for practical use could be achieved. We found not only that this is possible, but also that the performance and functionality obtained are more than sufficient for teaching system programming. Based on this encouraging result, we grew the prototype built for the experiment into a full-fledged learning environment, as described in this thesis. The original prototype still exists as one of the demos forming part of the jor1k project.[1]

The goal of this project is to develop a tool useful for teaching system programming in self-paced courses as well as in classrooms, and to demonstrate ways to improve how system programming is taught. Even though a lot of work still needs to be done, the tool developed so far, as described in the thesis, is fairly close to fulfilling this goal. We envision the project as a core part not just of CS 241, but of university-level system programming courses across the globe.

[1]`https://s-macke.github.io/jor1k/demos/compile.html`

REFERENCES

[1] "CS 492/493/494 Senior Projects," 2015, Department of Computer Science, UIUC. [Online]. Available: https://seniorprojects.cs.illinois.edu/

[2] "CS 241 (Fall 2015) Course homepage," 2015, Department of Computer Science, UIUC. [Online]. Available: https://courses.engr.illinois.edu/cs241/fa2015/

[3] L. Angrave, N. Gupta, S. Walters, E. Ahn, J. Tran, A. K. Singh, and S. Seth, "sysbuild v0.16.0," Nov. 2015. [Online]. Available: http://dx.doi.org/10.5281/zenodo.35683

[4] S. Macke, N. Gupta, B. Burns, J. Bölsche, G. Braad, J. Troelsen, S. Kristiansson, hak8or, J. Goense, Fabian, and E. M. Hvidevold, "jor1k dependency for sysbuild-v0.16.0," Apr. 2015. [Online]. Available: http://dx.doi.org/10.5281/zenodo.35684

[5] "CS 241 official course profile," 2015, Department of Computer Science, UIUC. [Online]. Available: https://cs.illinois.edu/courses/profile/CS241

[6] "Learn C — Free Interactive C Tutorial," 2015. [Online]. Available: https://www.learn-c.org/

[7] "Ideone — Online compiler and IDE," 2015. [Online]. Available: https://ideone.com/

[8] "Code Moo — A playful way to learn programming," 2015. [Online]. Available: http://www.codemoo.com/

[9] "Codecademy, free interactive code learning online," 2015. [Online]. Available: http://www.codecademy.com/

[10] "Code School," 2015. [Online]. Available: https://www.codeschool.com/

[11] "Khan Academy," 2015. [Online]. Available: https://www.khanacademy.org/

[12] L. Angrave, "System Programming wiki-book," 2015, Community-built Wiki. [Online]. Available: https://github.com/angrave/SystemProgramming/wiki

[13] "CS 241 (Fall 2015) MP7 — Wearables," 2015, Department of Computer Science, UIUC. [Online]. Available: https://courses.engr.illinois.edu/cs241/fa2015/mps/mp7/

[14] "CS 241 (Spring 2014) MP7 — Web Server," 2014, Department of Computer Science, UIUC. [Online]. Available: https://courses.engr.illinois.edu/cs241/sp2014/mp/mp7_skeleton/doc/html/

[15] "GitHub," 2015. [Online]. Available: https://github.com/

[16] "The Git version control system," 2015. [Online]. Available: https://git-scm.com/

[17] "The cs-education organization on GitHub," 2015. [Online]. Available: https://github.com/cs-education

[18] "cs-education/sysbuild on GitHub," 2015, cs-education. [Online]. Available: https://github.com/cs-education/sysbuild

[19] "cs-education/sysassets on GitHub," 2015, cs-education. [Online]. Available: https://github.com/cs-education/sysassets

[20] "cs-education/sys-staging on GitHub," 2015, cs-education. [Online]. Available: https://github.com/cs-education/sys-staging

[21] "cs-education/sys on GitHub," 2015, cs-education. [Online]. Available: https://github.com/cs-education/sys

[22] "cs-education/jor1k on GitHub," 2015, cs-education. [Online]. Available: https://github.com/cs-education/jor1k

[23] "The Yeoman scaffolding tool," 2015. [Online]. Available: http://yeoman.io/

[24] "The Yeoman "Webapp" Generator," 2015. [Online]. Available: https://github.com/yeoman/generator-webapp

[25] "The Bootstrap Framework," 2015. [Online]. Available: https://getbootstrap.com/

[26] "The Grunt JavaScript Task Runner," 2015. [Online]. Available: http://gruntjs.com/

[27] "The Bower package manager," 2015. [Online]. Available: http://bower.io/

[28] "The Sass CSS preprocessor," 2015. [Online]. Available: http://sass-lang.com/

[29] S. Macke and contributors, “s-macke/jor1k on GitHub,” 2015. [Online]. Available: https://github.com/s-macke/jor1k

[30] “GitHub Pages,” 2015. [Online]. Available: https://pages.github.com/

[31] “Improvements to GitHub Pages,” 2015, GitHub Blog. [Online]. Available: https://github.com/blog/1715-faster-more-awesome-github-pages

[32] “The jQuery library,” 2015. [Online]. Available: https://jquery.com/

[33] “The jQuery UI Layout plugin,” 2015. [Online]. Available: https://github.com/allpro/layout

[34] “The Knockout JavaScript library,” 2015. [Online]. Available: http://knockoutjs.com/

[35] “The Sammy.js web framework,” 2015. [Online]. Available: http://sammyjs.org/

[36] “The Ace embeddable code editor,” 2015. [Online]. Available: https://ace.c9.io/

[37] S. Macke and contributors, “jor1k,” 2015. [Online]. Available: http://jor1k.com/

[38] *OpenRISC 1000 Architecture Manual*, Architecture Version 1.1, OpenCores, Apr. 2014.

[39] S. Macke, “jor1k — Technical details,” 2015, Wiki. [Online]. Available: https://github.com/s-macke/jor1k/wiki/Technical-details

[40] S. Macke, “jor1k — Benchmark with other emulators,” 2015, Wiki. [Online]. Available: https://github.com/s-macke/jor1k/wiki/Benchmark-with-other-emulators

[41] “Google Analytics,” 2015. [Online]. Available: https://www.google.com/analytics/

[42] M. A. Tischer, “Improving the Assessment of Student Code,” ECE Undergraduate Senior Thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, May 2013, unpublished. [Online]. Available: http://hdl.handle.net/2142/47619