RustOS ASLR

Team 21

Peter Dong

1.
Goal

1. Goal

By the end of this course, I want to:

- Build up a full memory stack including the stack, heap, and other memory region with allocation functions such as malloc and free.
- Implement ASLR for the memory region of a process to make the layout randomized every time we reboot our Pi.

2. Motivation

2. Motivation

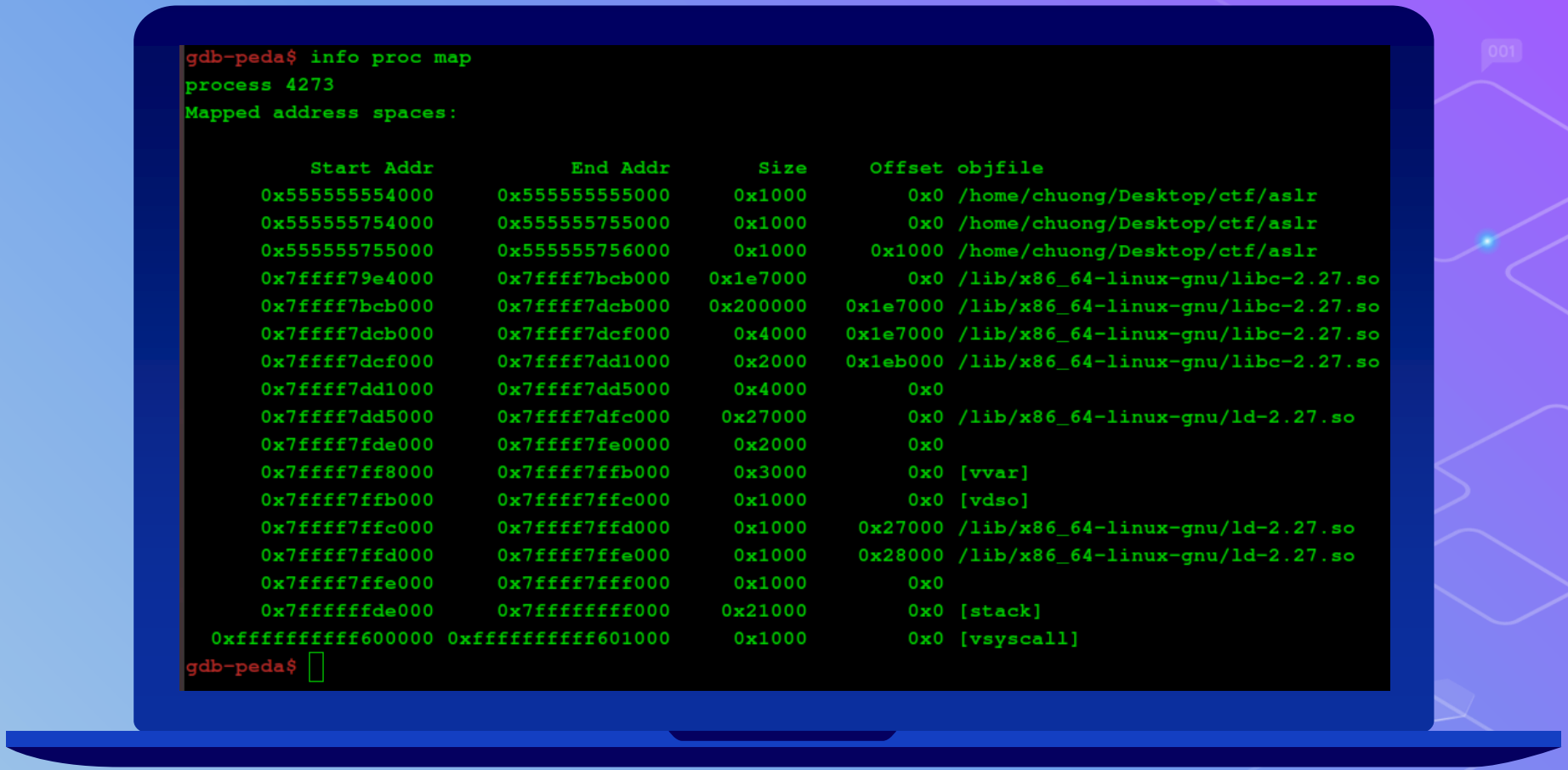
- ASLR is the one of the things I hate the most when doing CTF challenges, so I want to try and implement it to understand about it more.
- I want to make my project a bit security-related so I can flex about it during internship interviews.

3.
Demo Plan

3. Demo Plan

a) **Demoing the full memory stack**

- From what I'm imagining, each process would have its own stack and heap region
- I will try to demo allocating and deallocating objects in both these regions to show that I got the memory part to work
- I will also include functions to print out the information of a process's mapped address space!


gdb-peda$ info proc map
process 4273
Mapped address spaces:
Start Addr End Addr Size Offset objfile
0x555555554000 0x555555555000 0x1000 0x0 /home/chuong/Desktop/ctf/aslr
0x555555754000 0x555555755000 0x1000 0x0 /home/chuong/Desktop/ctf/aslr
0x555555755000 0x555555756000 0x1000 0x1000 /home/chuong/Desktop/ctf/aslr
0x7ffff79e4000 0x7ffff7bcb000 0x1e7000 0x0 /lib/x86_64-linux-gnu/libc-2.27.so
0x7ffff7bcb000 0x7ffff7dcb000 0x200000 0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
0x7ffff7dcb000 0x7ffff7dcf000 0x4000 0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
0x7ffff7dcf000 0x7ffff7dd1000 0x2000 0x1eb000 /lib/x86_64-linux-gnu/libc-2.27.so
0x7ffff7dd1000 0x7ffff7dd5000 0x4000 0x0
0x7ffff7dd5000 0x7ffff7dfc000 0x27000 0x0 /lib/x86_64-linux-gnu/ld-2.27.so
0x7ffff7fde000 0x7ffff7fe0000 0x2000 0x0
0x7ffff7ff8000 0x7ffff7ffb000 0x3000 0x0 [vvar]
0x7ffff7ffb000 0x7ffff7ffc000 0x1000 0x0 [vdso]
0x7ffff7ffc000 0x7ffff7ffd000 0x1000 0x27000 /lib/x86_64-linux-gnu/ld-2.27.so
0x7ffff7ffd000 0x7ffff7ffe000 0x1000 0x28000 /lib/x86_64-linux-gnu/ld-2.27.so
0x7ffff7ffe000 0x7ffff7fff000 0x1000 0x0
0x7ffffffde000 0x7ffffffff000 0x21000 0x0 [stack]
0xffffffffff600000 0xffffffffff601000 0x1000 0x0 [vsyscall]
gdb-peda$

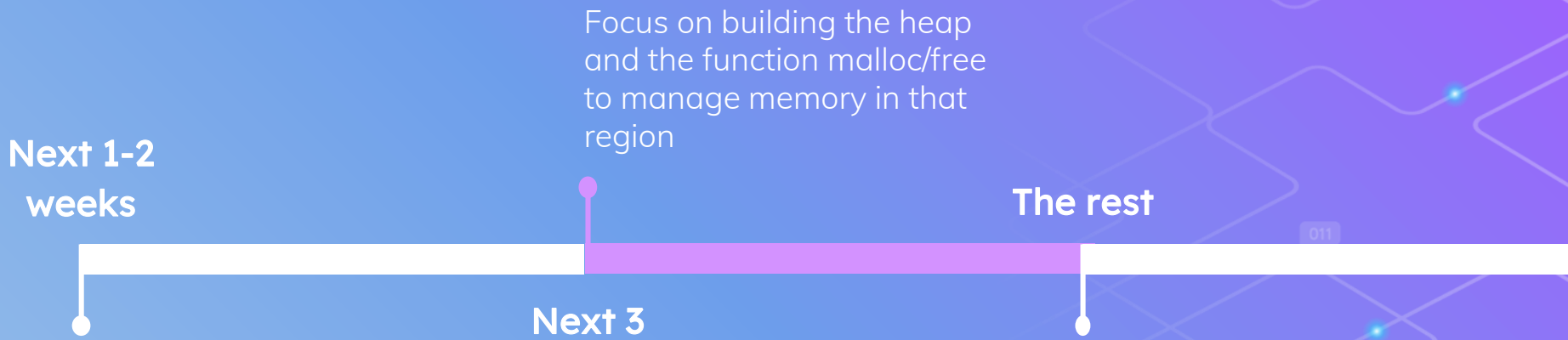
3. Demo Plan

b) Demoing ASLR

After implementing ASLR, it's not too hard to demo this. I can allocate something on the heap and print out its address in kmain. When I reboot the Pi, the address of this object should be randomized in the memory address space!

4. Timeline

4. Timeline


Next 1-2 weeks
Do more research on virtual memory; Try to integrate what we have so far to build up the heap and stack region for future project on address virtualization
Next 3 weeks
Focus on building the heap and the function malloc/free to manage memory in that region
The rest
Implement a randomize function to build up ASLR

Thanks!

Any questions?

Evaluate Team#21

Next: Team#22

Next to next: Team#24

RustOS HDMI Capability Framebuffer to HDMI

Team 22: Jonathan Buchanan, Tanner Jones, Kunal Mitra

Objective

- Want to be able to utilize the HDMI port!
- Create drivers for the GPU
- Be able to display our RustOS shell onto any monitor through the HDMI
- Also to be able to display various images.

Timeline

1. Develop mailbox driver (March 7th)
2. Implement video core driver (March 14th)
3. Display shell (March 28th)
4. Display an image (April 9th)

Extensions

- Display jpeg image on monitor saved in file systems
- Play mp4 video on monitor saved in file system

Evaluate Team#22

Next: Team#24

Next to next: Team#25

Supporting User-Level Threads
Team 24

Aditya Jituri, Kelly McMaster, Ida Wang

Overview of Objective

- Support user-level threads

- Support stdlib's concurrency

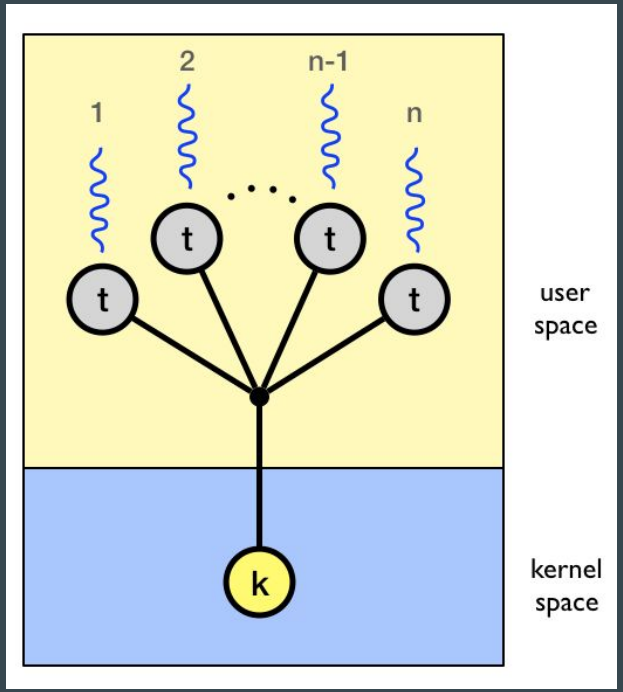


Figure 1. Many User Threads to One Kernel Thread Model

Source: Uppsala University

Modularity

- Baseline (Minimum Viable Product):
  - Many-to-one, no support for multiple kernel threads
  - No SW support for multiple cores
- If we finish early:
  - Implement some form of kernel-level multithreading
    - Switch to the one-to-one model
  - Add support for the multiple cores on the Raspberry Pi 3

Challenges

- Lack of knowledge of multi-thread programming
- Need more research understanding of user-level threads
- Concurrency

Timeline

By March 1: Research + Design

By Pi Day (3/14): Prototype the many to one user-level threads (have internal demo by mid-March)

By April Fools Day (4/1): Create MVP (Many to One User Level Threads)

By 4/20: Demo and fine-tune

References

Thread Implementation Models

Available: https://cs.brown.edu/research/thmon/thmon2a.html

Implementing Threads:

Available: http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Concurrency in C++

Available: https://www.c-sharpcorner.com/article/programming-concurrency-in-cpp-part-1/

Difference Between User Level and Kernel Level Threads

Available: https://stackoverflow.com/questions/15983872/difference-between-user-level-and-kernel-supported-threads

Evaluate Team#24

Next: Team#25

Next to next: Team#26

ASLR in RustOS

Bradley Dover and Wil Warner

What is ASLR?

- Address space layout randomization
- Protects against buffer overflow attacks
- Works with virtual memory to randomize the location of program components on every run
- Removes ability for attackers to reliably jump to exploited functions

Why ASLR?

- Ensure the safety of our Pi
- Prominently used in operating systems today (iOS, Windows, Linux, MacOS)
- Learn how to combat cyber crime

Project Breakdown

- Phase 1: Generate and store randomized offsets for program components on program launch
- Phase 2: Ensure that processes can access their randomized memory regions using the offset information
- Phase 3: If time permits, implement KASLR

Evaluate Team#25

Next: Team#26

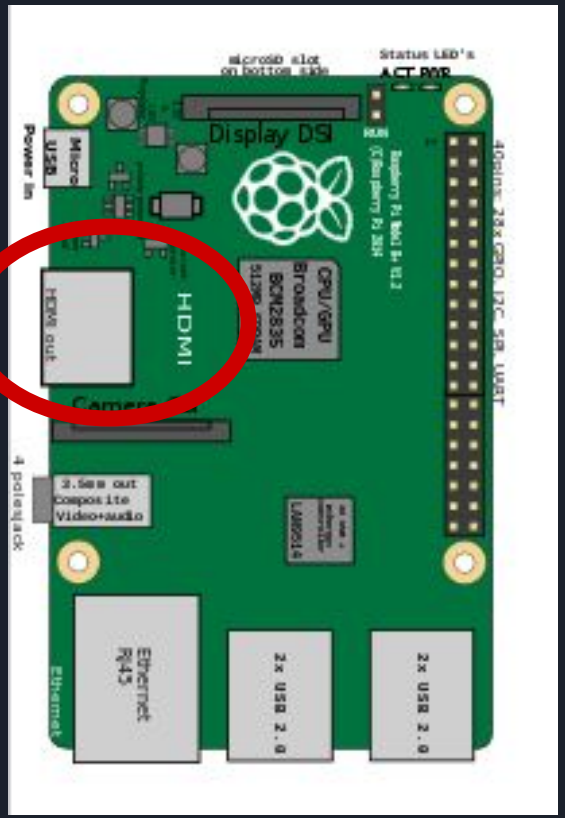
Next to next: Team#30

Operating Systems Final Project : HDMI Output and Keyboard Input

Tucker, Will, and Camille

How — Hardware

- Configurable HDMI port
- Write to HDMI output via Framebuffer



How — Software

- Code we need to change:
  - Output information to a buffer the HDMI can interpret rather than printing using a laptop's terminal interface.
- Software we need to add:
  - Handling the PI framebuffer (through a series of 'Mailbox' registers)
  - Handling the PI HDMI configuration

Why this project

- Implementing external device drivers is an important part of all operating systems.
- HDMI makes use of DMA and the PI's Framebuffer.

Evaluate Team#26

Next: Team#30

Next to next: Team#33

CS 3210 Final Project Proposal

Connor Truono, Davy Waku, Sean Walsh

The Problem

- As part of lab 3, we are implementing a memory allocator, drivers for the Raspberry Pi's EMMC, and a FAT32 / vFAT filesystem for our toy RustOS
- Powerful but limited in functionality - the filesystem is read-only to make the assignment more streamlined
- A true operating system needs to be able to write to the file system as well. Without this core feature, our RustOS is extremely limited in real-world functionality

Our Proposed Solution

- In order to make our RustOS toy operating system feature complete, we propose modifying the file system to allow for write access to the user
- Write-access should allow the user to perform three operations - creating, modifying, or deleting a file, or some combination of these actions (e.g. a ‘move’)
- This should be done in a memory-safe way that takes into account and prevents the corruption of the existing contents of the file system at the time of modification
- Our implementation should also preserve the ability for the current file system to be accessed in read-only mode
- The user should be able to perform these actions using standard commands like, *touch*, *mv*, *rm*, etc. from our custom shell

Deliverables

1. Implement the ability for RustOS to mount its FAT32 file system in either read-only mode (its current implementation) or in read-write mode

2. In read-write mode, add onto the existing file system to allow for users to manually create, delete, and modify files while protecting the integrity of existing contents in memory

3. Create standard terminal commands that the user can use to write to and modify the file system

Deliverable 1 - File System Mountable in Different States

- Allow user to select which mode (r or rw) they want to use to mount the file system
  - Either through an explicit command
  - Or through configuration files

Deliverable 2 - Memory-Safe Writeable File System

- Our project would prepare an interface by which users or other code could write files to mounted file systems
- Editing, copying, writing, deleting, and moving (or some subset of these features, depending on time constraints) would be implemented
- Implementation would be done in a memory-safe way that does not overwrite existing files

Deliverable 3 - Terminal Commands

- If our RustOS is to be feature complete, there needs to be a way for the user to dynamically modify the file system
- Just like how we implemented standard UNIX commands like *cd*, *ls*, *pwd*, etc. in lab 3, we must implement terminal commands to allow the user to perform all core writeable actions (create, modify, delete, etc.)
- We would like to recreate the following terminal commands, as they provide full functionality:
    - *touch* - the ability to create a new file
    - *rm* - the ability to delete a file
    - *mv* - the ability to move or rename a file
    - *sed* - A stream editor to modify a file. *Vi* could be supported to do this as well, but filtering / adding onto a file stream is a more realistic undertaking given our time frame

Timeline

- Now - March 2 (hopefully): Finish lab 3
  - The read-only file system lab gives the knowledge of file systems needed to manipulate FAT32 for writability
  - There's no use in making a writable file system if we haven't built the read-only version first
- March 2 - March 22: Figure out requirements
  - Make an outline for all the structures needed
  - Divide into smaller tasks
- March 23 - April 8: Make it happen!
  - Start programming to develop at least an incomplete version of every deliverable before presentation days
  - Revise scope of the project as necessary
- April 9-April 20: Finishing touches and Submission

Evaluate Team#30

Next: Team#33

Next to next: Team#34

Audio Driver support in Raspberry Pi 3 using Rust

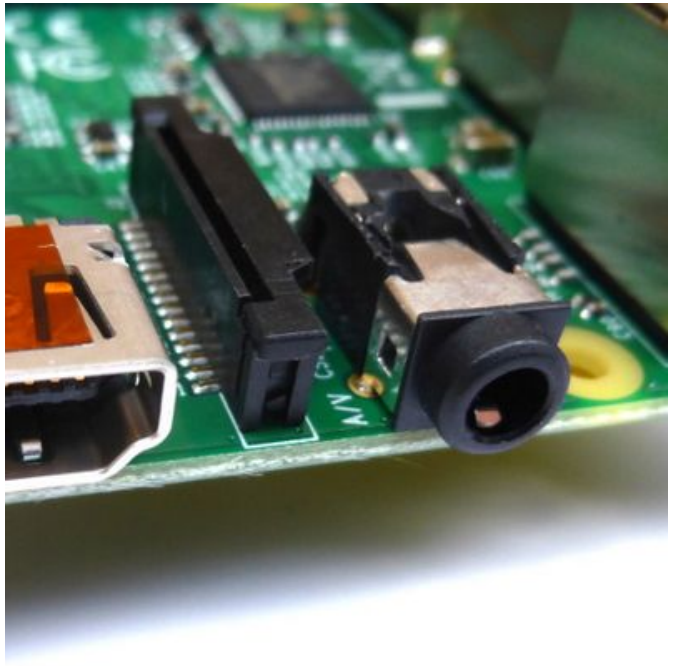
Team 33: Peyton McGill

Reasoning for Implementation

- Majority of operating systems today make use of audio feedback to let users experience a more responsive OS (include notifications of errors, process finishing, even media consumption)
- A high number of applications developed for a system will be able to make use of the provided functionality.
- The audio driver will allow for use of devices such as headphones and stereos with the pi

Challenges and Goals

- By the end of the project there will be support for audio playback via the Raspberry Pi's 3.5mm audio jack.
- User will be able to play a number of sounds through shell commands and decompressed audio files are able to be read off of the SD card reader.
- Begins with correctly writing code that correctly implements PWM (pulse width modulation) using pins of the board.
- Possibility of more complex file support (compressed audio, mp3, who knows?)



Demonstration

This driver proves simple enough to demonstrate by just showing multiple sounds of different pitch that the user can access with the OS shell. Speakers will be brought in to play music off of multiple types of files.

Timeline

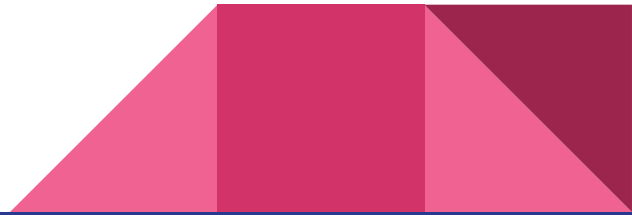
< March 2: Finish lab 3 study for Quiz 1

By March 8: Research and begin work on the code that will allow for use of PWM on the board with Rust

By March 30: PWM audio conversion of frequencies completed user commands either completed or in the works (Lab 4 also done hopefully)

By April 5: Raw audio files are able to be played and users commands finished.

Till demo date: Bug fix



Evaluate Team#33

Next: Team#34

Next to next: Team#35

Process Scheduling

Abhijeet Saraha

Proposal Idea

- Implement several different scheduling policies similar to linux scheduler, multi-level feedback queues, shortest remaining time etc.
- Develop benchmarks to test/assess the different scheduling policies.
- Explore something similar to a hybrid scheduler that changes policies given a specific workload.

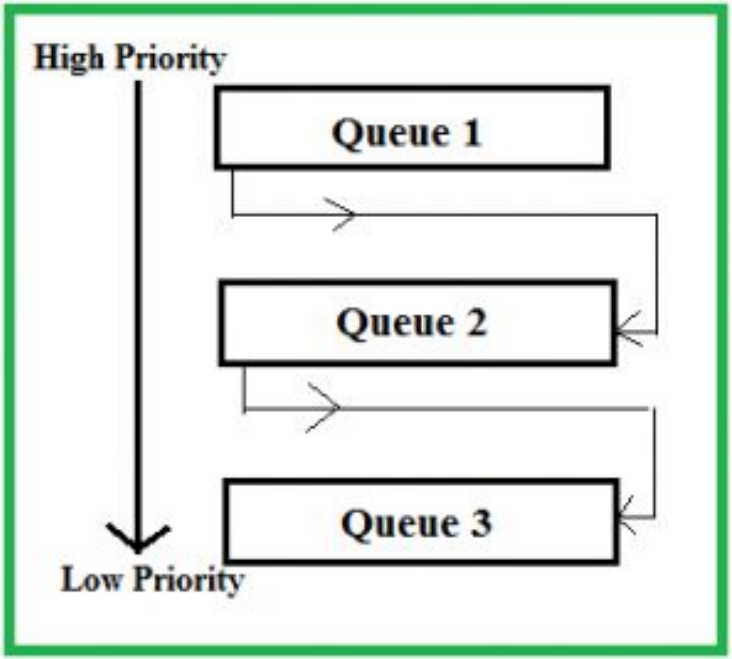
Objective

- Run benchmark tests using round robin policy to establish a baseline.
- Measure metrics such as:
  - Throughput
  - Wait time
  - CPU utilization etc.
- Run the same benchmarks using the new policies.
- Evaluate the performance of the scheduler compared to round robin benchmark.

Timeline

- Week 1: Research different scheduling policies and implementations.
- Week 2: Develop benchmarks and test the scheduling implementations.
- Week 3: Debug any problems so far and explore the possibility of a hybridized scheduler based on benchmark results.
- Week 4: Clean the repo and prepare the demo.

Implementation



Evaluate Team#34

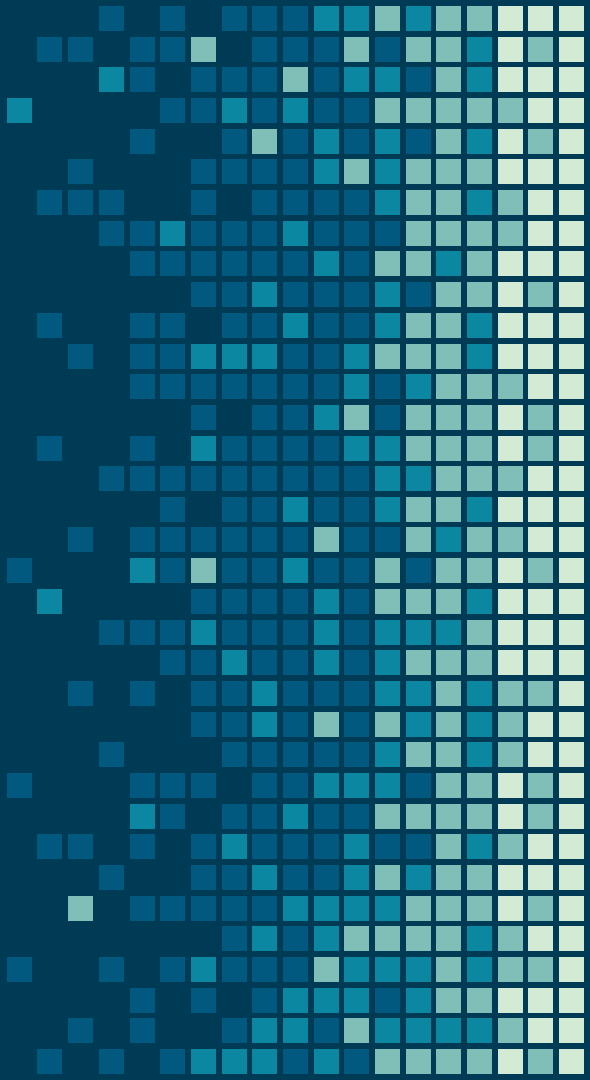
Next: Team#35

Next to next: Taesoo Kim

Driver for FPGA Bitcoin Mining

Team #35

Joey Kilpatrick

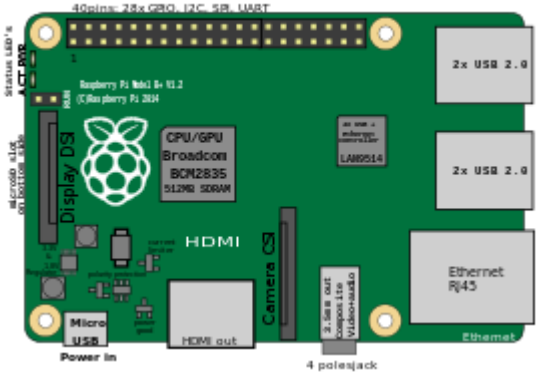


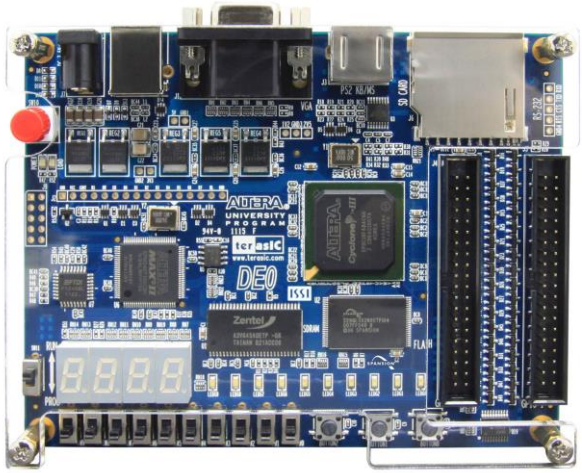
Background

- Bitcoin mining relies on the **SHA-256** hashing algorithm.
- Computers in the network race to find a value that, when hashed, produces a known hash.
- **Mining steps (simplified):**
  - Take a random 256-bit value.
  - Hash the number using SHA-256
  - Check if the hash matches the desired value
    - If yes, done.
    - If no, repeat.
- The process is designed to be resource intensive and calculating a single hash can take hundreds of CPU cycles.
- Mining is usually done on **custom hardware** that can perform these hashes much faster and more efficiently.

Project

- **Write a driver for custom hashing hardware on an Altera Cyclone V FPGA on a DE0-CV board.**
  - Driver will be written in Rust and will run on Raspberry Pi
  - Driver will communicate with FPGA through GPIO





Timeline

- Will use an **iterative development approach**, building up the API
- There will be **two deliverables per phase**:
  - Verilog code for the FGPA
  - Rust code for the device driver
- **Phase 0: GPIO Overhead** *(By March 17)*
  - IO for both devices for receiving and sending data through GPIO
- **Phase 1: Hashing API** *(By March 24)*
  - Send data and receive the hash of that data from FPGA
- **Phase 2: Mining API** *(By April 9)*
  - Send desired hash and FPGA hashes until a pre-image is found.

Demonstration Goals

- To be able to demonstrate the Phase 1 Hashing API
- To be able to demonstrate the Phase 2 Mining API
- To compare the runtime of the FPGA against a software implementation of the mining algorithm on the Raspberry Pi's CPU.

Evaluate Team#35

Next: Taesoo Kim