ASLR & DEP

DAVID HEAVERN

Problem Statement

- Security Issues
  - Buffer Overflows
    - Buffer overflows are the source of many of the common vulnerabilities in modern software
    - Caused by not checking the source length when writing to a buffer
  - Smashing the stack
    - Overruns on the stack start overwriting data higher up in the stack
    - Overwriting the EIP can allow an attacker to hijack the program flow on a functions return to execute malicious code
  - Shellcode Injection
    - EIP could be redirected to the stack itself where the attacker has written malicious machine code and this code will be executed

Idea

JOS ASLR – Address space layout randomization

- Randomizes the memory space locations upon program execution
- Locations of the stack, heap, and shared libraries are less predictable
- Makes hijacking the EIP less dangerous by lowering odds of success

JOS DEP – Data Execution Prevention

- Prohibits parts of memory that hold data from being executed
- Marks pages like the stack and the heap unexecutable
- Prevents shellcode injection in the stack or the heap to prevent malicious code execution

Demo Plan

Attack Mitigation Scenarios

- Highlight vulnerable aspects of common programming idioms
- Exhibit different types of attacks on vulnerable systems without ASLR or DEP
- Step through the attack with ASLR and DEP and show how attacks are mitigated
- Discuss other security vulnerabilities that are still possible

Timeline

March 28th - April 1st

- Research alternatives and Evaluation

April 4th – April 8th

- ASLR Implementation

April 11th – April 13th

- DEP Implementation

April 13th – End of Semester

- Demo preparation
- Extra time?
  - Shared library support

386 Emulator

Stephan Williams

Problem Statement

- Implement a simple 80386 emulator in Rust sufficient to boot JOS.

Idea

- Need to implement:
  - Barebones BIOS
  - Real and Protected mode
  - Subset of 16- and 32-bit assembly (as determined by examining compiler output)
  - Disk and Display I/O
  - Virtual Memory
- Maybe:
  - Segmentation
  - Memory Protection
  - Keyboard I/O
- Stretch goals:
  - Interrupts

Demo Plan

- Be able to boot JOS at least to the point where it can display the initial prompt

Timeline

- Week 1:
  - Basic program architecture
  - Some sort of BIOS
  - Disk I/O
  - Implement instructions as needed
- Week 2:
  - More instructions
  - Virtual Memory
- Week 3:
  - More Instructions
  - Display I/O

Linux Kernel Module Driver for Keyboard LEDs

Bridging the Gap Between the Kernel Space and the User Space:

Connor Reeder

Problem Statement:

The driver which Linux currently uses to activate and deactivate the Caps Lock and Num Lock LED lights on a Toshiba Satellite C55-A5286 currently does not allow for control from user space applications. The functionality of those two lights is bound to the standard functions of Caps Lock mode and Num Lock mode, respectively. Thus, there is no way to repurpose the lights to serve other functions in the event that the user does not use those lights for their current function.

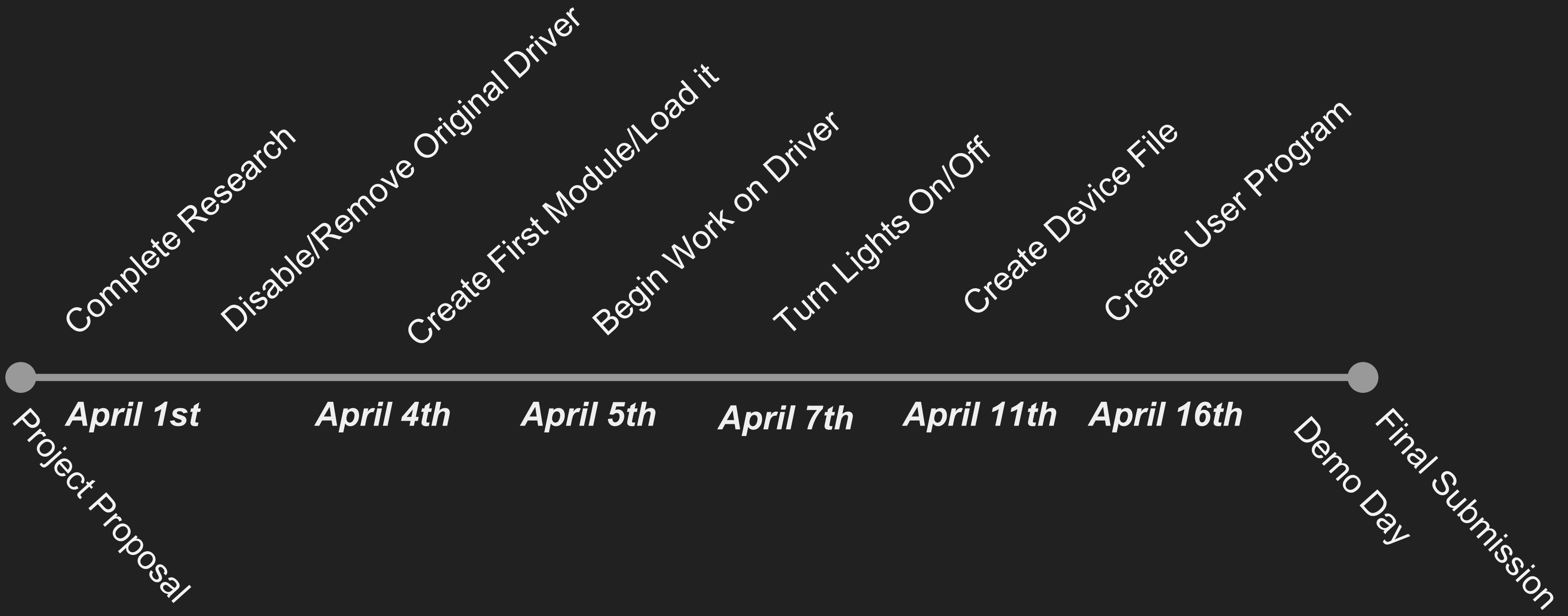
Idea

- Write a Linux kernel module which containing a driver for the Toshiba Satellite C55-A5286 keyboard LED lights which will replace the one currently in use.
- The driver will mount each of the two LED lights as a linux special file node in the /dev directory so as to allow any user space application to read and write to it like any other device.
- It will be mounted as a character device, thus requiring applications to read and write to it in block-aligned sizes.
- Create a simple user space program which will use the caps lock light as a notification for some type of event.

Demo Plan

- Break down the procedure that was used to develop the linux kernel module, the risks that were involved, and what exactly it does to carry out its task.
- Show how the interface works between the kernel space module and the user space application, including the device files /dev/capslight and /dev/numlight, and the Linux commands necessary to develop, install, or remove a module.
- I will demonstrate lighting with echo {0,1} > {/dev/capslight,/dev/numlight} and reading with cat {/dev/capslight,/dev/numlight} in a bash script that blinks a light when any terminal command fails.
- Present the documentation or specifications that were researched in order to understand the specifics of how to control the LED lights built into the Toshiba Satellite C55-A5286.

Timeline


Complete Research
Disable/Remove Original Driver
Create First Module/Load it
Begin Work on Driver
Turn Lights On/Off
Create Device File
Create User Program
Project Proposal
April 1st
April 4th
April 5th
April 7th
April 11th
April 16th
Demo Day
Final Submission

Transactional Synchronization Extensions in JOS

•••

Robby Guthrie

What is Transactional Memory?

Alternative to software locks

Sequences of loads and stores to memory are committed in an atomic transaction

Allows parallel access to data structures provided the same memory regions are not accessed

Ex. Concurrent Hash Maps

What is Intel TSX?

Set of instructions to support memory transactions starting in Haswell microarchitecture

XBEGIN, XEND mark beginning and end of transaction regions

Intel hardware monitors for multiple threads of control for conflicts

A conflict is two threads accessing memory in the same cache line

Plans for using Intel TSX in JOS

A fork of QEMU provides TSX support: http://www.cs.berkeley.edu/~sltu/papers/tsx.pdf

My goal: replace locking mechanisms in JOS with transactional memory where appropriate

1.) Set up facilities to benchmark the JOS kernel

2.) Replace the global kernel lock with fine-grained access

3.) Compute the speedup of JOS with TSX vs. primitive locking mechanisms

Potential Problems

Benchmarking on a simulator will likely be inaccurate, especially if TSX is performed in software

Discovering regions of JOS code where TSX is likely to provide speedup. A lot of global data structures are linked lists which don't support random access.

Timeline

Over Spring Break: Have JOS running in QEMU+TSX

First two weeks of April: Have facilities for benchmarking JOS kernel code in place

Final: Speed up JOS with TSX as much as possible.

Implement ASLR

Jiateng Xie

Problem Statement

1. Buffer overflow is one of the most commonly seen attacks. To protect the system against it, ASLR (address space layout randomization) is used.
2. Basically, it happens when a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations.

Problem Statement

- voidhello()
- {
- charvul1[8]="This";
- charvul2[8];
- gets(vul2);
- printf("vul1:%s\nvul2:%s\n",vul1,vul2);
- }

- The program is vulnerable because the the input vu[2] may be larger than 8 bytes and overwrite previous memory locations.
- Attackers may also transfer the control of the program by inserting well-written shellcode, e.g., ret2libc attack.

Idea

- To prevent attackers from jumping to specific parts of memory by overflowing buffers, the solution is to arrange the address space of key data areas randomly, so that the attacker can't do it reliably.

Demo Plan

- To show that a well-crafted buffer overflow attack will work when ASLR is turned off, and it will not work when it is turned on. And also show that the address space of the process is arranged randomly by printing out the addresses.

Timeline

- One member team, so I will try my best to get the previously mentioned functionality to work before the deadline.

CS 3210 Project Proposal

Porting JOS to 64-bit

Saikrishna Arcot

March 16, 2016

Background

- Taken CS 2200 (Systems and Networks) and CS 4290 (Advanced Computer Architecture)

Background

- Taken CS 2200 (Systems and Networks) and CS 4290 (Advanced Computer Architecture)
- Aware of some of the memory differences between 32-bit and 64-bit architectures

Proposal

Port JOS to run as a 64-bit kernel on a x86-64 CPU

Goals

- Compile JOS as a 64-bit kernel

Goals

- Compile JOS as a 64-bit kernel
- Run JOS on a x86-64 CPU

Stretch Goals

- Be able to compile the kernel in both 32-bit and 64-bit mode

Stretch Goals

- Be able to compile the kernel in both 32-bit and 64-bit mode
- Add/verify support for using more than 4 GB of memory

Stretch Goals

- Be able to compile the kernel in both 32-bit and 64-bit mode
- Add/verify support for using more than 4 GB of memory
  - The previous steps should allow for using/accessing more than 4 GB of virtual memory, but having more than 4 GB of physical memory might need some work.

Stretch Goals

- Be able to compile the kernel in both 32-bit and 64-bit mode
- Add/verify support for using more than 4 GB of memory
  - The previous steps should allow for using/accessing more than 4 GB of virtual memory, but having more than 4 GB of physical memory might need some work.
  - Current BIOS code appears to support only up to 64 MB of memory, even though more is provided by QEMU.

Stretch Goals

- Be able to compile the kernel in both 32-bit and 64-bit mode
- Add/verify support for using more than 4 GB of memory
    - The previous steps should allow for using/accessing more than 4 GB of virtual memory, but having more than 4 GB of physical memory might need some work.
    - Current BIOS code appears to support only up to 64 MB of memory, even though more is provided by QEMU.
- Anything else I can think of or is suggested

Demo

- Demonstrate that the JOS kernel boots up as a 64-bit kernel

Demo

- Demonstrate that the JOS kernel boots up as a 64-bit kernel
- Demonstrate user programs run correctly

Demo

- Demonstrate that the JOS kernel boots up as a 64-bit kernel
- Demonstrate user programs run correctly
- Demonstrate that the kernel can still be compiled and run as a 32-bit kernel (if stretch goal is met)

Demo

- Demonstrate that the JOS kernel boots up as a 64-bit kernel
- Demonstrate user programs run correctly
- Demonstrate that the kernel can still be compiled and run as a 32-bit kernel (if stretch goal is met)
- Demonstrate that more than 4 GB of physical memory can be used (if stretch goal is met)

Timeline

1. Research what changes are necessary at the bootloader level to support a 64-bit kernel (about 2-3 days)

Timeline

1. Research what changes are necessary at the bootloader level to support a 64-bit kernel (about 2-3 days)
2. Make changes to the assembly files for loading and starting the kernel (about 3 days)

Timeline

1. Research what changes are necessary at the bootloader level to support a 64-bit kernel (about 2-3 days)
2. Make changes to the assembly files for loading and starting the kernel (about 3 days)
3. Make changes to C files related to virtual addresses (about 2 days)

Time Sensitive Scheduling

By Brian Surber and Darrington Altenbern

Problem Statement

- Modern applications often have processes with realtime deadlines
- This spans processes from audio/video encoding to control inputs on a plane.
- Without specific actions by the scheduler, these processes can miss their deadlines in favor of fairness towards less sensitive processes.

Idea

- The idea is to implement a time sensitive scheduler. This means adding to process structs to tell the scheduler about deadlines and expected execution times.
- The goal is to minimize deadline misses instead of maximizing fairness for all processes.
- The scheduler itself would still operate with a priority-based round robin algorithm, but would make exceptions for processes that could potentially miss their deadlines if not given resources.
- A separate process would be executing in the background of the OS, predicting the need to give resources to the time sensitive processes over standard processes.

Idea cont.

- The background process could predict deadline misses multiple steps ahead and begin special resource allocation early.
- Priority would be dynamically set for processes that need the resources and be used to set order of executing without damaging the round-robin scheduling.
- Expected execution times (saved in the process structs) can be dynamically adjusted based on feedback from the completed processes.
- When choosing between multiple time sensitive processes, multiple algorithms will be tested: probably shortest-first or first-in-first-out.

Demo Plan

+ The metric to be analyzed for each selection algorithm is deadline hit/miss ratios.
+ This can be simulated with various benchmark tests where each time-sensitive process can add to a miss or a hit counter based on its success.
+ Comparisons will be done between our existing JOS scheduler, a time-sensitive scheduler using FIFO to determine high priority process order, and a time-sensitive scheduler using SF to determine process order.

Timeline

- The main focus will be on implementation of at least one of the time-sensitive scheduling algorithms (SF or FIFO).
- After at least one is implemented, the group can choose to implement another or move onto developing benchmarks.
- If extra time is available, additional benchmarks can be created to simulate edge cases or high stress situations.
  - i.e. very short execution time processes adding to scheduler overhead

Extending JOS to include Networking Capabilities

•••

By:
Zain Rehmani
Adithya Nott

Problem Statement: What is this project about?

- Implementing network support in JOS
    - transmitting/receiving packets
- Expanding on initial network driver support in Lab 6 in more interesting ways via challenge problems and/or DSM

Why is adding a network driver to JOS helpful?

- The world is more connected than ever
- Interaction with other devices like routers is present in modern Operating Systems
- Without networking drivers, good luck accessing the Internet.
- Solid opportunity to explore the Application and Transport Layers of Networking.
- To extend the capabilities of JOS as far (and farther) as the MIT 6.828 curriculum prescribed

Distributed Shared Memory

- Distributed Shared Memory (DSM) system that applies the networking drivers by connecting multiple computers together to share memory
    - Separated physical memory all treated as one address space
- Page faults handled across a network of computers
- Applies multiple concepts: Networking, Multi-threading, Paging

Plan + Timeline

- Finish core Lab 6 a week before Lab 5 is due. (April 4)
- As Lab 6 is being finished, determine direction based on feasibility (complete at least two things by April 11)
    - Challenge problems already proposed in Lab 6
        - "Add a simple chat server to JOS, where multiple people can connect to the server and anything that any user types is transmitted to the other users..."

    - Distributed Shared Memory (DSM) between two computers
        - Use sockets to communicate between computers for DSM
            - DSM initialization process is considered part of this.
        - mprotect()
        - DSM Page fault handler function
        - Implement threading library for DSM
            - This is a stretch goal; using pthreads instead until this is attempted

Demo Plan

- Show that all the default checks pass with ./grade-lab6
- Demo any finished functionality (DSM paging interactions, web chat interface, etc.) between two laptops simultaneously running JOS.
    - If DSM functionality hasn't reached this level, between two processes run on one laptop.

Any Questions, Comments, or Suggestions?

JOS : GUI

•••

JESSE LEE
YEONJOON CHOI

Problem Statement

- JOS environment lacks Graphical User Interface. In the modern operating system, graphical user interface is one of the main component that differentiate Operating systems. Therefore, window manager is a crucial component of the operating system that we need to know how to design.

Idea

1. Implement a simple GUI for JOS that has the following functionalities / components
   a. Components
      i. Mouse {1}
      ii. Window {2}
   b. Functionalities
      i. Shutdown {3}
      ii. Restart {4}
      iii. Access Terminal {5}

*priorities are denoted inside {}

Demo Plan

- “Make qemu” will call the GUI of the JOS kernel
- The interface will have the following features
  - Implementation of Mouse Events (Click, Visible mouse pointer)
  - Shutting down with 1-click
  - Access to the kernel

Timeline

1. Understand the xv6-public code
2. Start implementation on *init.c* to boot our GUI from the JOS kernel
3. Draw one pixel to the JOS window
4. Draw cursor
5. Draw the window
6. Get “onclick” function working (mouse_events)

PAGING TO DISK

Sneh Munshi

Bhavani Jaladanki

Problem Statement

- **Problem:** Virtual memory space is limited!
- **Solution:**
    - *Page Swapping:*
        - Program will be able to use any page it wants regardless of whether the page is already in memory or not.
        - Swapping should replace a page in memory that will not be used in the near future, with the page in the disk that the program wants.
- **Goal:** make sure that the page switched out is one that is rarely used since permanent storage is slower than memory

Idea (Part 1)

- **Paging Server (Component 1):**
  - Server (instead of disk)
    - Server will take in 3 IPCs: page in, page out, and delete page
    - *Page Out:*
      - Server will get IPC that specifies which page needs to be sent out from memory into the disk
      - Using bitmap, find empty block on hard drive and write page's contents to block.
      - 20-bit index of the disk block that the page was written to will be sent out within an IPC
    - *Page In:*
      - server will take the 20-bit index given to it to read the page from the disk into the memory
      - page will be shared to environment that asked for the page
      - block on the disk will be freed using the bitmap.
    - *Discard Page:*
      - block on the disk will just be freed using the bitmap

Idea (Part 2)

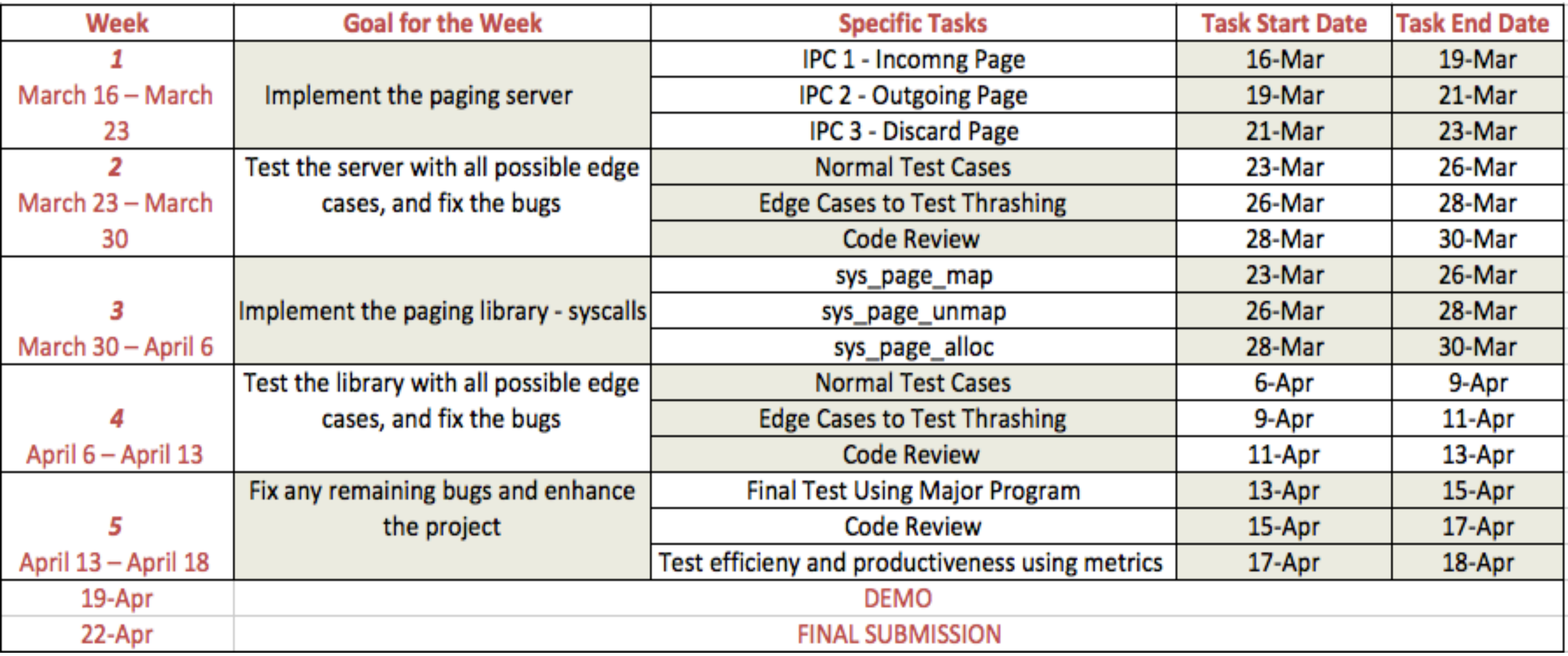
- **Paging Library (Component 2):**
  - Using exo-kernel style library:
    - Make sure that paging done in a proper way
    - Page-faults that arise from trying to access a page that has been swapped out is also properly handled by syscalls
      - Some syscalls are:
        - *sys_page_map*: maps page, but if mapping a page that's paged to disk, bring back into memory, and if we want to map over a page that is paged to disk, the IPC of delete page is sent.
        - *sys_page_unmap*: same as sys_page_map, except unmaps isntead
        - *sys_page_alloc:* returns –E_NO_MEM when no more memory to allocate page, so paging starts (pages go out to disk in order to make more memory space)

Demo Plan

- We will hopefully handle all the test cases to make paging work, and we will present these test cases in class
- Some test cases might include:
  - Page in: handling page fault for accessing page on disk (so bring the page in successfully to memory)
  - Page out: swap a page in memory back to disk
- Actual Demo Steps:
  - Pick a program that requires a lot of memory space
  - Show that the program cannot run on JOS because it will eventually run out of memory in trying to hold all the pages it needs
  - Then show that the program can run on our modified OS since page swapping is a feature that will make sure the program has all its required pages.

Timeline

- There are **two** major components to this project: the paging library and paging server
- We have **5 weeks** until the demo



Week	Goal for the Week	Specific Tasks	Task Start Date	Task End Date
1 March 16 – March 23	Implement the paging server	IPC 1 - Incomng Page	16-Mar	19-Mar
		IPC 2 - Outgoing Page	19-Mar	21-Mar
		IPC 3 - Discard Page	21-Mar	23-Mar
2 March 23 – March 30	Test the server with all possible edge cases, and fix the bugs	Normal Test Cases	23-Mar	26-Mar
		Edge Cases to Test Thrashing	26-Mar	28-Mar
		Code Review	28-Mar	30-Mar
3 March 30 – April 6	Implement the paging library - syscalls	sys_page_map	23-Mar	26-Mar
		sys_page_unmap	26-Mar	28-Mar
		sys_page_alloc	28-Mar	30-Mar
4 April 6 – April 13	Test the library with all possible edge cases, and fix the bugs	Normal Test Cases	6-Apr	9-Apr
		Edge Cases to Test Thrashing	9-Apr	11-Apr
		Code Review	11-Apr	13-Apr
5 April 13 – April 18	Fix any remaining bugs and enhance the project	Final Test Using Major Program	13-Apr	15-Apr
		Code Review	15-Apr	17-Apr
		Test efficieny and productiveness using metrics	17-Apr	18-Apr
19-Apr	DEMO			
22-Apr	FINAL SUBMISSION			

Evaluation

- **Efficiency:**
  - We will check to see that there is not a very high rate of page outs and page ins (check ratio of page ins to page outs). Basically, make sure that the number of disk accesses can be as low as possible.
- **Productiveness:**
  - Say a program allocated lots of pages. Then, a small program should not have to keep swapping pages if it needs to allocate memory even though it's a small program.

CS 3210: Final Project
Remote Procedure Call

Ning Wang
David Benas
Zongwan Cao

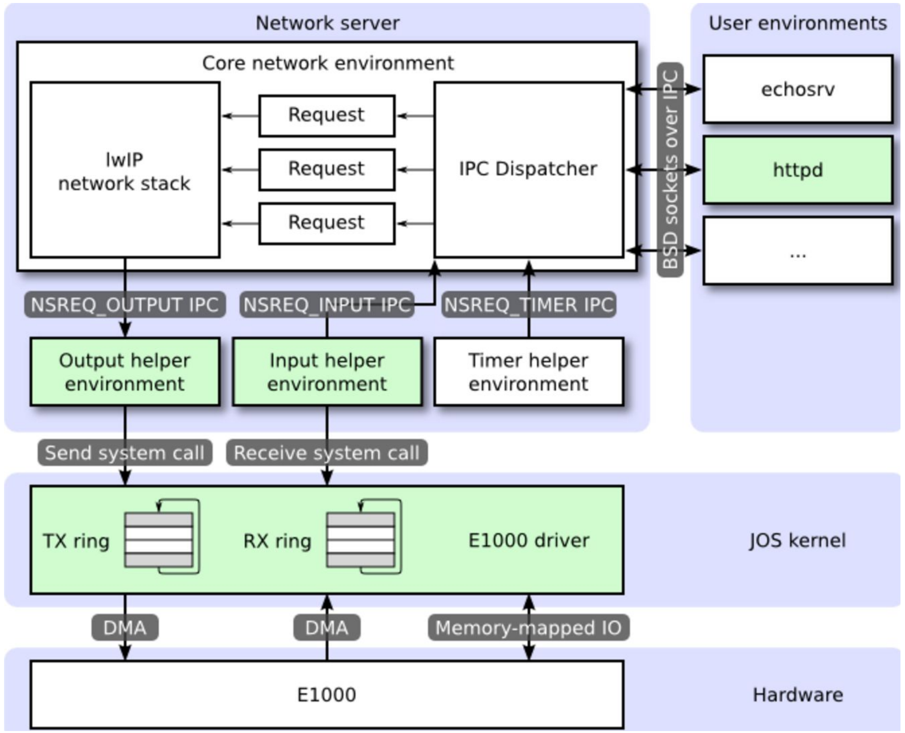
Problem Statement

- Socket Interface is hard to use
  - Procedure call is a well understood mechanism
  - The JOS kernel can be extended to support remote procedure call (RPC)
  - It provides a very useful paradigm for communication across network
  - It also makes it easier to build distributed systems
- But...Currently JOS does not have network support
  - No self respecting OS should go without a network stack
  - We need to build a network driver before implementing RPC

Part I: Network Driver

- QEMU has support for virtual network
    - It simulates E1000 network interface card
    - We need to write a driver for E1000
- We also need a network stack
    - Write a TCP/IP protocol stack from scratch is hard work
    - We use lwIP, a lightweight protocol stack suite
    - It runs in user space and implement a BSD socket interface

Part I Architecture



Part II: RPC

- Two ways to implement RPC
  - We can build on top of the socket interface or directly talk to kernel network buffer
  - We need to write the stub code for both client and server to hide the network
  - We also need to write the actual client and server code to transmit the packets
- Potential issues
  - How to design the interfaces
  - What is the semantics in terms of communication failures
  - How to marshal arguments into network packets
  - How to reduce the communication overhead

Demo Plan

- Part I demo:
  - Show the functional test cases and underlying architecture of the Lab 6 code.
  - Demonstrate what was added to JOS and how it contributes to our solution of the problem statement.
- Part II demo:
  - There are two potential ways to demonstrate RPC
    - Write a chat server using the RPC library and demonstrate it
    - Write a distributed key-value store using the RPC library and demonstrate it

Timeline

Week of:	March 14	March 21	March 28	April 4	April 11	April 18
Proposal	🟧					
Spring Break & Initial research		🟩				
Lab 6 & Prerequisite lab(s)			🟧	🟧		
RPC Implement & Proof of Concepts				🟩	🟩	
Debugging & Just-In-Case Time					🟧	
Final Deliverable Creation					🟩	🟩

Q & A

Porting JOS to the ARM architecture

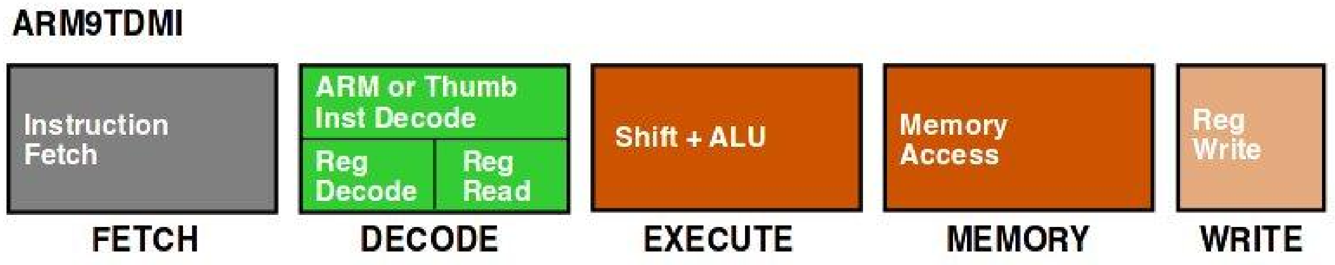
CS 3210 Final Project Spring 2016

Problem Statement

- This project aims to implement basic functionality of JOS on the Raspberry Pi using the ARM ISA
- Motivation
  - New ISA to learn, used in mobile/embedded systems
  - Understand what makes an OS portable

ARM ISA - Background

- **RISC**
- **Operating modes** = { User, FIQ, IRQ, Supervisor/System, Abort, Undefined }
- All instructions **conditionally executable**
- Pipeline (5-stage)
- **Full descending stack** (Stack grows down, SP is lowest occupied location)
- Registers = **30 general purpose**, 1 PC, 6 “Program Status” registers
- Registers **allocated by mode**: Mode A implies Mode B-F “banked” (saved)



General Registers and Program Counter Modes

User32	FIQ32	Supervisor32	Abort32	IRQ32	Undefined32
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

Program Status Registers

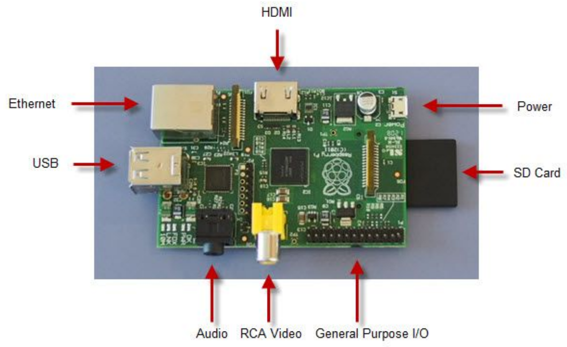
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

Hardware challenge

- We will try to boot JOS on a Raspberry Pi 1 Model B, which features a Broadcom 2835 SoC.

- The hard part is going to be loading the kernel into memory. The BIOS is proprietary but it does the job of loading the bootloader.

- In the end if we succeed we will demo the Rpi through a projector. Perhaps some other interesting things like memory-mapped I/O.



Project - Goals/Demo Plan

Lab 1, 2, 3 Goals

- Show JOS booting to monitor
- Memory initialization functions using ARM’s view of memory - show memory tests succeeding
- Show interrupts and tests for user environments succeeding

Lab 4 +

- Multiprocessing support

Project - Timeline

Prerequisites

- Cross-compilation environment
- Reading

Lab 1, 2, 3 Goals

- Bootloader (~1 week)
- Memory management (~½ week)
- User environments/interrupts (~½ week)

Lab 4 +

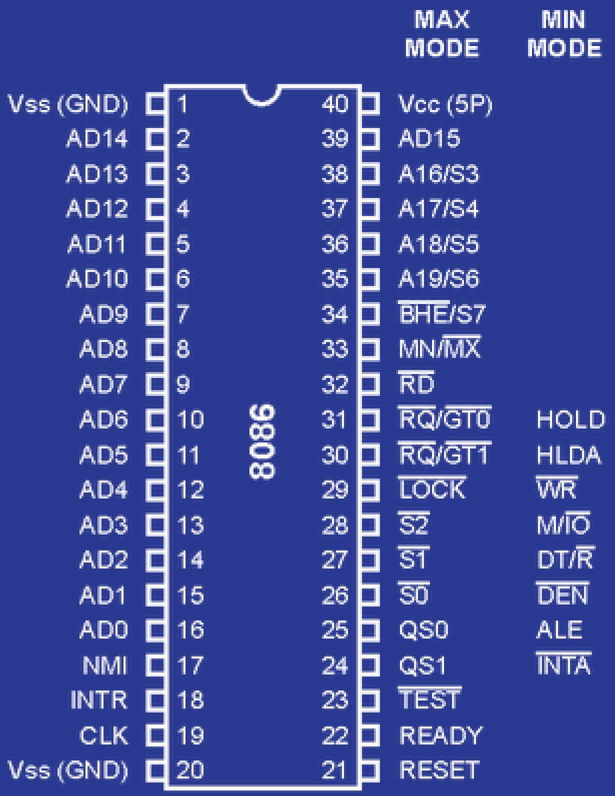
- Reach goal

Proposal: Intel 8086 Emulator

By: Daniel Engel
Erick Lin
Thomas Shields

Problem Statement

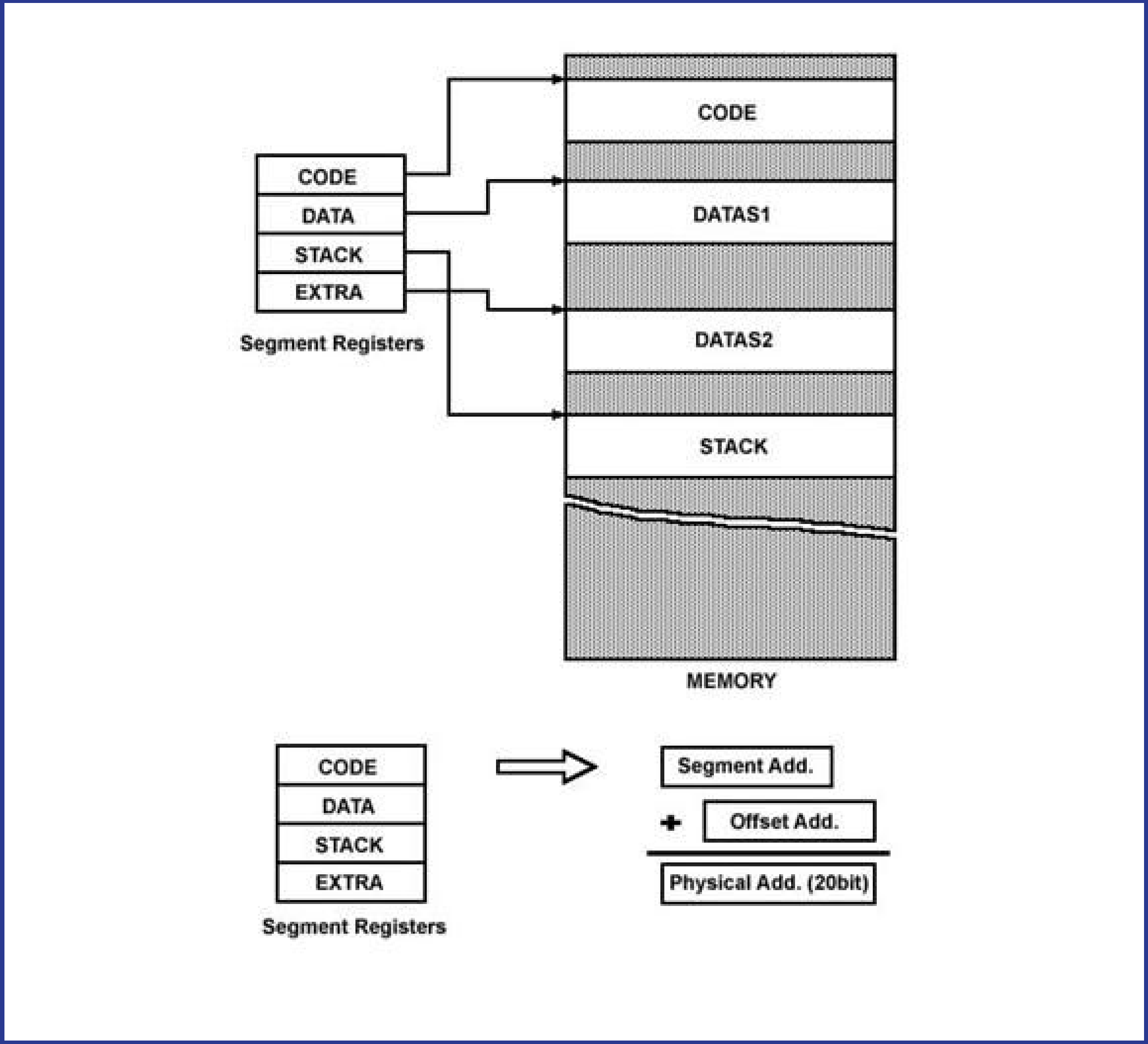
- To emulate the core functionality of the Intel 8086 processor and have it run legacy software such as DOS programs



Intel 8086 Description

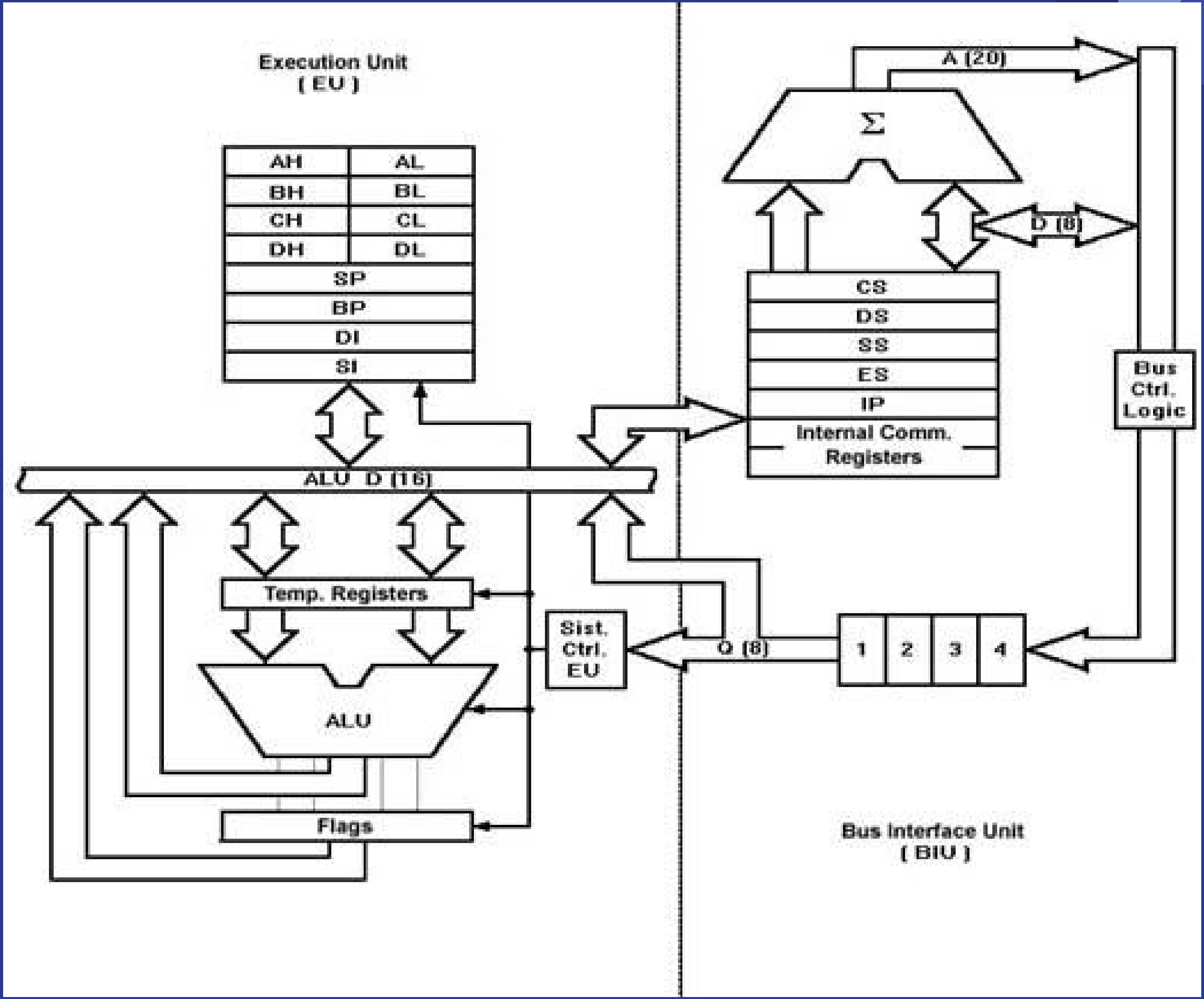
- Little-endian
- No virtual memory
- Uses segmented memory – segments are 16 bits and physical addresses are 20 bits
- The data bus size and registers are 16 bits but memory is byte-addressable
- The CPU architecture is divided into an execution unit (EU) and a bus interface unit (BIU)
- Dedicated locations in memory for processing system interrupts and RESET function

Segmented Memory



EU and BIU

- EU and BIU have separate instruction pipelines which communicate with one another
- EU wait mode caused by branches, memory accesses, and instructions that require many clock cycles


Execution Unit
(EU)
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
DI
SI
ALU D (16)
Temp. Registers
ALU
Flags
Sist. Ctrl. EU
Q (8)
1
2
3
4
A (20)
Σ
D (8)
CS
DS
SS
ES
IP
Internal Comm. Registers
Bus Ctrl. Logic
Bus Interface Unit
(BIU)

Timeline

- Implement memory, registers, and the CPU with simple instruction pipelining
- Write parser and assembler for 8086 assembly to DOS executable formats (e.g. COM, MZ)
- Emulate the hard disk and system interrupts (Priority Interrupt Management Controller)
- Support graphics
- Support audio if enough time

Demo Plan

- Ideally, we would be able to show some examples of legacy software being run on our emulator.

Questions?

Final Project Proposal Extending JOS by Implementing mmap( )

Daniel Carnauba, Nicolette Fink, Thomas Coe

Problem Statement

- ★ read() and write() can be inefficient for non-sequential file access
    - ○ Lots of system call overhead (seeking)
- ★ Multiple processes accessing the same file can be inefficient
    - ○ Each process reads the file into a buffer in its individual memory space
- ★ Context switching can be costly when making many kernel calls for reading or writing files

Implementation Idea: mmap( ) and munmap( )

```
void *mmap(void *addr, size_t length, int prot, int flags,
                int fd, off_t offset);
```

mmap() creates a new mapping in the virtual address space of the calling process.

```
int munmap(void *addr, size_t length);
```

The **munmap**() system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references.

Implementation Idea: Mapping Modes

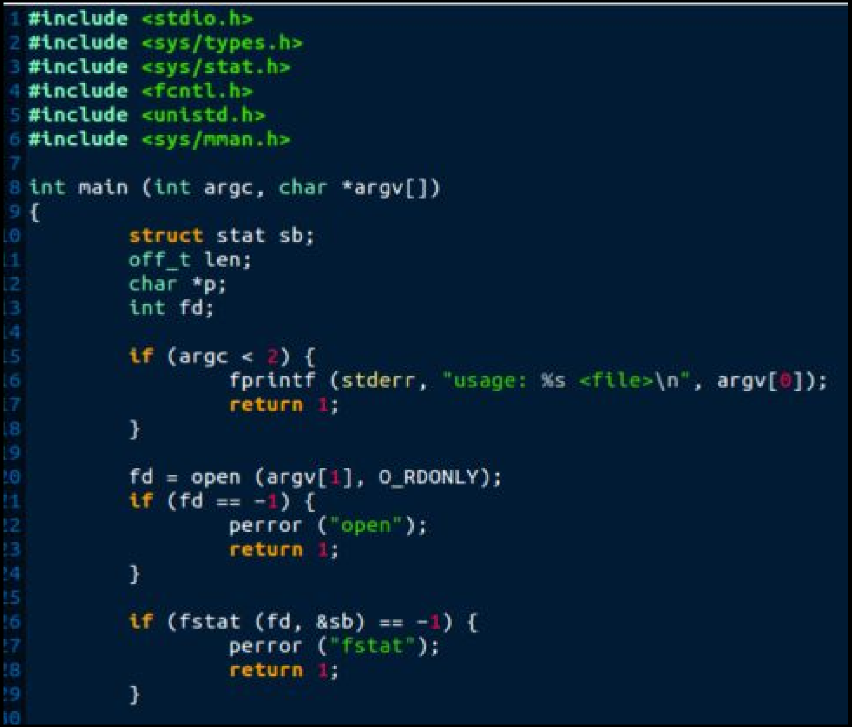
MAP_SHARED

Share this mapping. Updates to the mapping are visible to other processes that map this file, and are carried through to the underlying file.

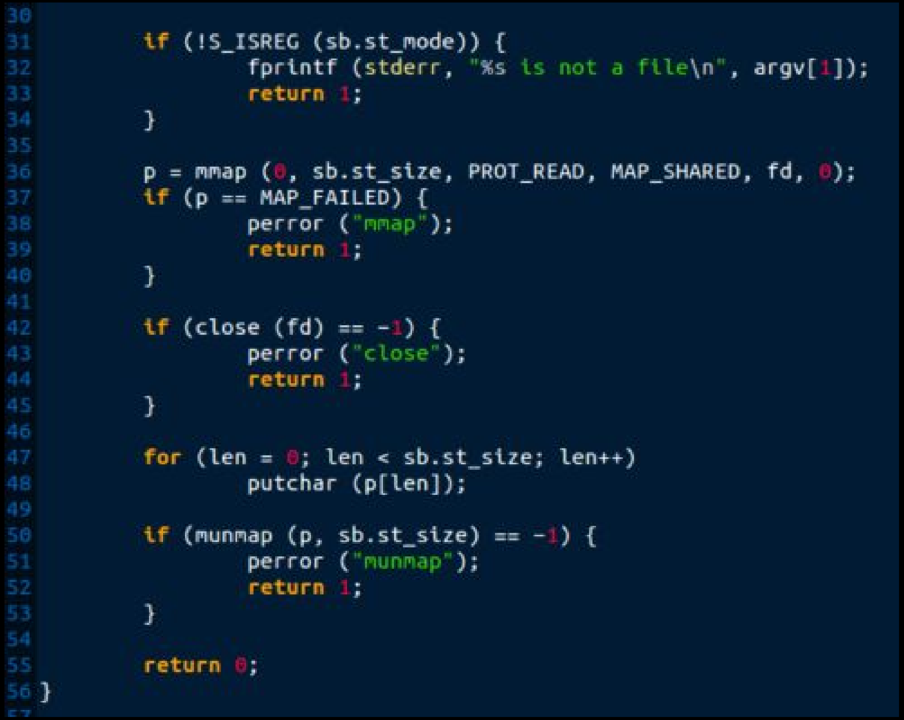
MAP_PRIVATE

Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the **mmap**() call are visible in the mapped region.

Mapping Example

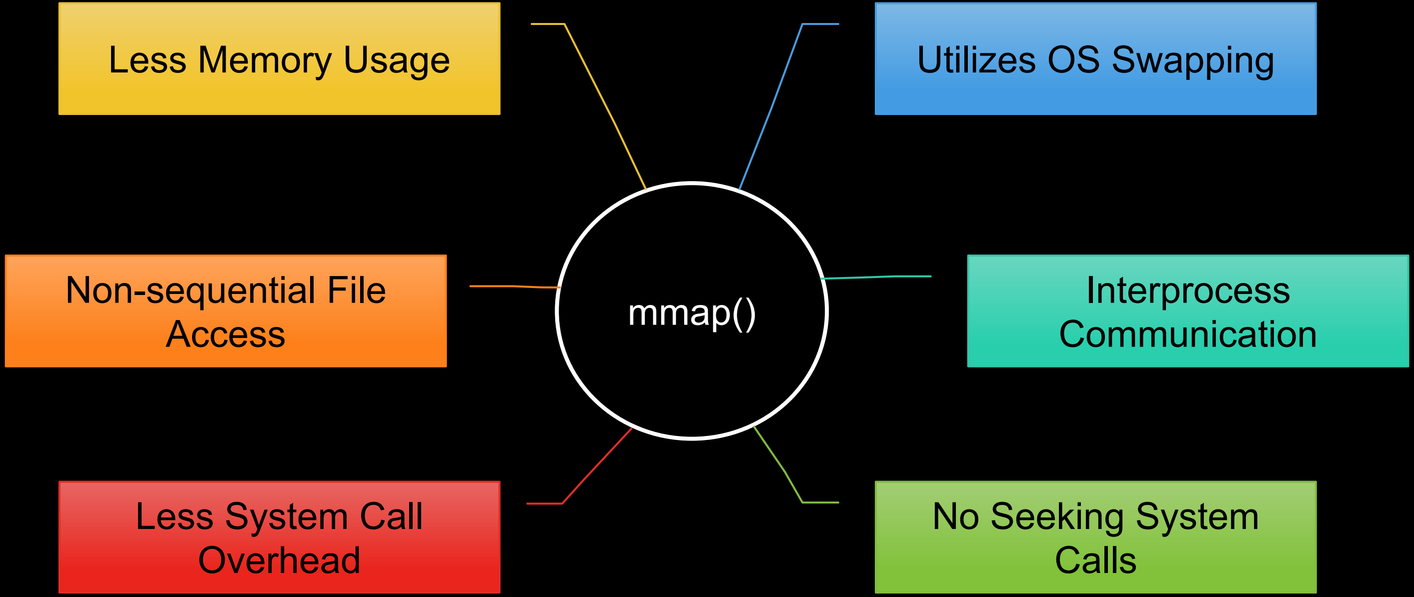


```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main (int argc, char *argv[])
{
        struct stat sb;
        off_t len;
        char *p;
        int fd;

        if (argc < 2) {
                fprintf (stderr, "usage: %s <file>\n", argv[0]);
                return 1;
        }

        fd = open (argv[1], O_RDONLY);
        if (fd == -1) {
                perror ("open");
                return 1;
        }

        if (fstat (fd, &sb) == -1) {
                perror ("fstat");
                return 1;
        }

```



```c

        if (!S_ISREG (sb.st_mode)) {
                fprintf (stderr, "%s is not a file\n", argv[1]);
                return 1;
        }

        p = mmap (0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
        if (p == MAP_FAILED) {
                perror ("mmap");
                return 1;
        }

        if (close (fd) == -1) {
                perror ("close");
                return 1;
        }

        for (len = 0; len < sb.st_size; len++)
                putchar (p[len]);

        if (munmap (p, sb.st_size) == -1) {
                perror ("munmap");
                return 1;
        }

        return 0;
}
```

Implementation Benefits


Less Memory Usage
Utilizes OS Swapping
Non-sequential File Access
mmap()
Interprocess Communication
Less System Call Overhead
No Seeking System Calls

Demo Plan

- ★ Benchmark mmap() vs. read()/write()
- ★ Create benchmark tests using a file searching application

- ★ For MAP_PRIVATE, compare:
  - ○ Number of system calls made
  - ○ Time spent running the tests
- ★ For MAP_SHARED, compare:
  - ○ Memory overhead

Timeline

Spring Break	Mar 22-Apr 2	Apr 3-Apr 9	Apr 10-Apr 16
Research & Initial Design	Final Design & Implementation	Implementation	Debugging & Final Touches

Suggestions?

ASLR

Brandon Jackson | Elliott Childre

Problem

Vulnerabilities such as a buffer-overflow allows an attacker to change the flow of execution to a memory address of their choosing.

The memory address space is predictable, making it easier for an attacker to jump to the location they want.

The attacker can jump to code that was injected into memory or an already existing shared library (libc).

Problem

```
[20:33:29]-ubuntu-(~) $ ldd demo
    linux-gate.so.1 =>  (0xb7785000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75bb000)
    /lib/ld-linux.so.2 (0x800e9000)
[20:33:30]-ubuntu-(~) $ ldd demo
    linux-gate.so.1 =>  (0xb7790000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75c6000)
    /lib/ld-linux.so.2 (0x8002b000)
```

Turn off ASLR:

```
[20:33:31]-ubuntu-(~) $ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
0
[20:33:56]-ubuntu-(~) $ ldd demo
    linux-gate.so.1 =>  (0xb7ffe000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7e34000)
    /lib/ld-linux.so.2 (0x80000000)
[20:33:58]-ubuntu-(~) $ ldd demo
    linux-gate.so.1 =>  (0xb7ffe000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7e34000)
    /lib/ld-linux.so.2 (0x80000000)
```

Problem

demo.c

```c
#include<stdio.h>
#include<stdlib.h>

void print(char *string)
{
    char buffer[50];
    strcpy(buffer, string);
    puts(buffer);
}

void main(int argc, char **argv)
{
    print(argv[1]);
    exit(0);
}
```

Printing the maximum amount without over-writing…

```
$ ./demo `perl -en 'print "A" x50'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA
```

Printing too much (writing over return address)

```
$ ./demo `perl -en 'print "A" x70'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAA
Segmentation fault (core dumped)
```

Opening a shell (ret2libc...)

```
gdb$ r `perl -e 'print "A"x62 . "\x90\x31\xe5\xb7" . "\xe0\x61\xe4\xb7" . "\x24\x3a\xf7\xb7"'`
Starting program: /home/brandon/demo `perl -e 'print "A"x62 . "\x90\x31\xe5\xb7" . "\xe0\x61\xe4\xb7" . "\x24\x3a\xf7\xb7"'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA�1���a��$:
[New process 28110]
process 28110 is executing new program: /bin/dash
[New process 28111]
process 28111 is executing new program: /bin/dash
$
```

Idea

Implement ASLR (Address Space Layout Randomization) which randomizes memory segments.

This makes memory addresses harder to predict, therefore harder to jump to.

Keep user code static but randomize the stack.

Effectiveness depends on the amount of space the segment could be placed in.

Demo Plan

Show that a user cannot execute injected code on that environment’s stack.

When running the same program multiple times, the user stack should start at a slightly different address each time.

Show that an attacker cannot brute-force guess a memory location. (sufficient entropy)

Schedule

1st week - Planning

2nd week - Randomizing Stack

3rd week - Randomizing kernel code (.text)

4th week - Remaining implementation and testing

5th week - Testing, preparing for demo, write-up

Triplicate

Demsar, Teeny, Brennick

Problem

- Data storage devices can be flaky.
- Corruption can occur silently.
  - Cosmic rays!
  - Fluctuating power supplies, physical shock and vibrations, and electromagnetic interference.
  - Relatively rare, but often enough for it to happen to you.

Proposal

- Automatically fix block-level errors that occurred due to silent corruption from external sources using data redundancy.

Detecting errors

- Mathematics
- Checksums can be calculated for blocks of data.
- CRC
    - Cyclic-redundancy-checksum
    - Fast and simple compared to cryptographic hashes.

Correcting errors

- From backups
  - Have to run manually, recovered manually. Hard to do with individual block errors. Would need to restore the entire file.
  - “Don’t tell me about a problem if you can fix it.”
- From redundant blocks
  - Somewhat complex to implement.
  - Needs to automatically recover corrupted blocks from good blocks.
  - Hurts performance since every block written to the disk needs to be written twice.
  - Cache is unaffected, so the performance hit shouldn’t be too bad.

Demo (test)

1. Corrupt a block on the IDE device using a corrupter kernel monitor function.
2. Show the data at the corrupted block and compare it to the still good block.
3. Read the file that contains the corrupted block.
4. Show that the corrupted block has been recovered with the contents of the good block.
5. Corrupt both blocks.
6. Kernel panic when the block cannot be recovered.

Implementation

- bio.c:bwrite
    - This is where buffered / cached blocks are flushed to disk.
    - We'll have to write the buffered block and a duplicate.
- bio.c:bget
    - Cached blocks are okay.
    - Reading blocks from disk, have to verify checksums and attempt to take corrective action
- Can test our modifications with xv6's stressfs.

Timeline

- 1 week - Implement checksum methods, look into creating demo file.
- 2 weeks - Write kernel monitor file corrupter.
- 3 weeks - Set up new block metadata structs, reserve new space for redundant blocks and checksums.
- 4 weeks - Change methods for editing block data to copy changes to redundant blocks, change block reading methods to perform checksum checks and attempt to recover from corruption.

Stretch Goals

- Fully mirror a cluster of IDE devices for redundancy and failover, at the filesystem level.
- Convert xv6’s journaling filesystem to a log-structured one.
- Option to control number of redundant blocks for greater or less redundancy

Paging to disk

by: Prem Saravanan, Henry Peteet, Millad Asgharneya

Problem Statement

With JOS now supporting multiple processes consider that we have 2 processes that use 1.5GB of RAM each.

Currently JOS has no way of dealing with needing more than 2GB of memory for all user-space programs and will either kill the process that pushes it over this limit, or double-fault depending on implementations.

Idea (changing how page faults are handled)

Out of memory (got -E_NO_MEM)

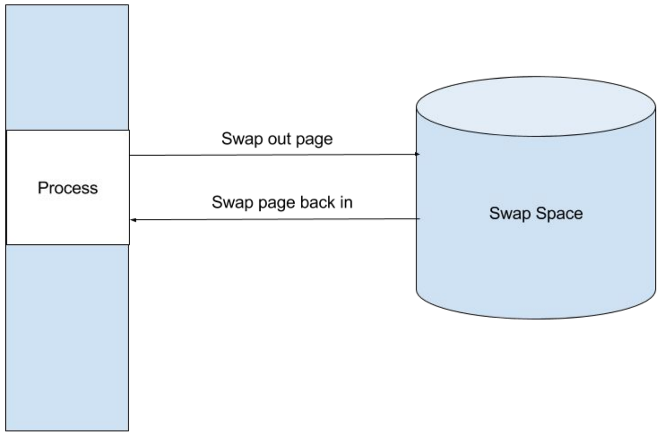
1. Mark page as evicted in the page table
2. Write victim frame to swap partition and mark the page as evicted
3. Allocate the evicted physical frame to the page table entry that faulted

Page was previously swapped out

1. Find an empty page or select a new victim page to swap
2. Restore the page from disk into the new page

Demo plans

1. Show JOS (traditional) failing to run 2x 1.5GB processes (using malloc repeatedly or some trivial example)
2. Show that these processes can now run under our version of JOS



Timeline

March 28
Finish Lab 4 (we are using lab 4 as our base JOS)

April 4
Read from disk and write to disk

April 11
Implement store_page and restore_page

April 11
Implement basic FIFO logic that can pick a page to be evicted

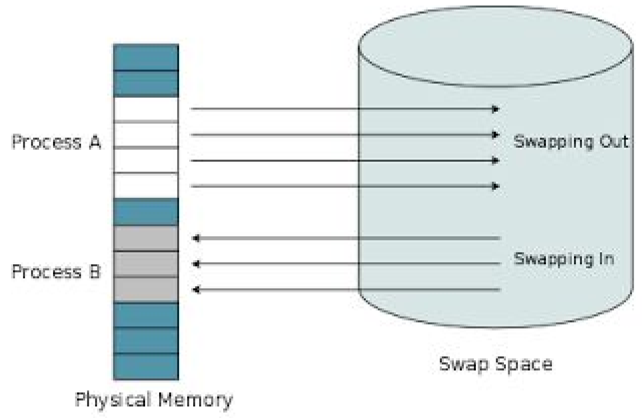
April 14
Combine parts 1-4

April 19-21
Demo day

Bonus

If we have extra time we would like to attempt the following alternatives to our eviction logic.

1. Shared swap space
2. LRU approximation with 2 bits
3. Evicting pages from other processes
4. Metrics for each page replacement policy

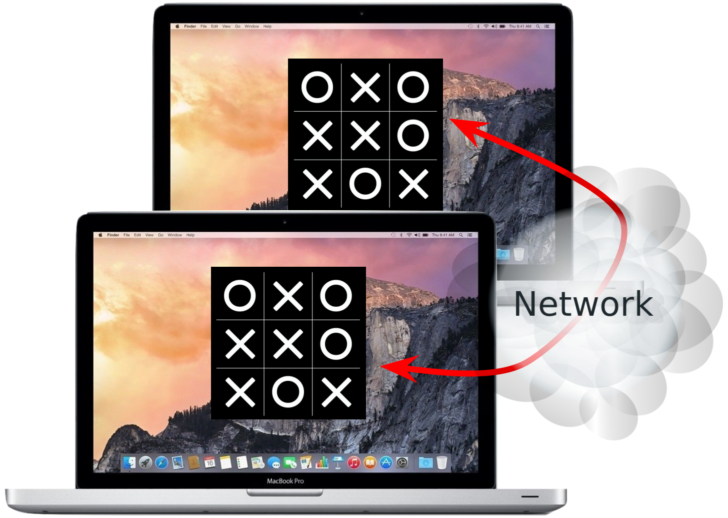


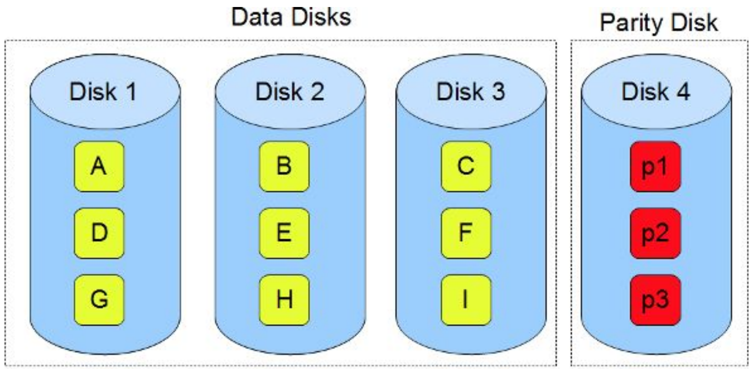
Final Project Presentation

Online Tic-Tac-Toe Game + RAID

Problem Statement

Online Tic-Tac-Toe interactive gaming plus **RAID** enabled hard drive.





RAID 4 – Blocks Striped. (and Dedicated Parity Disk)

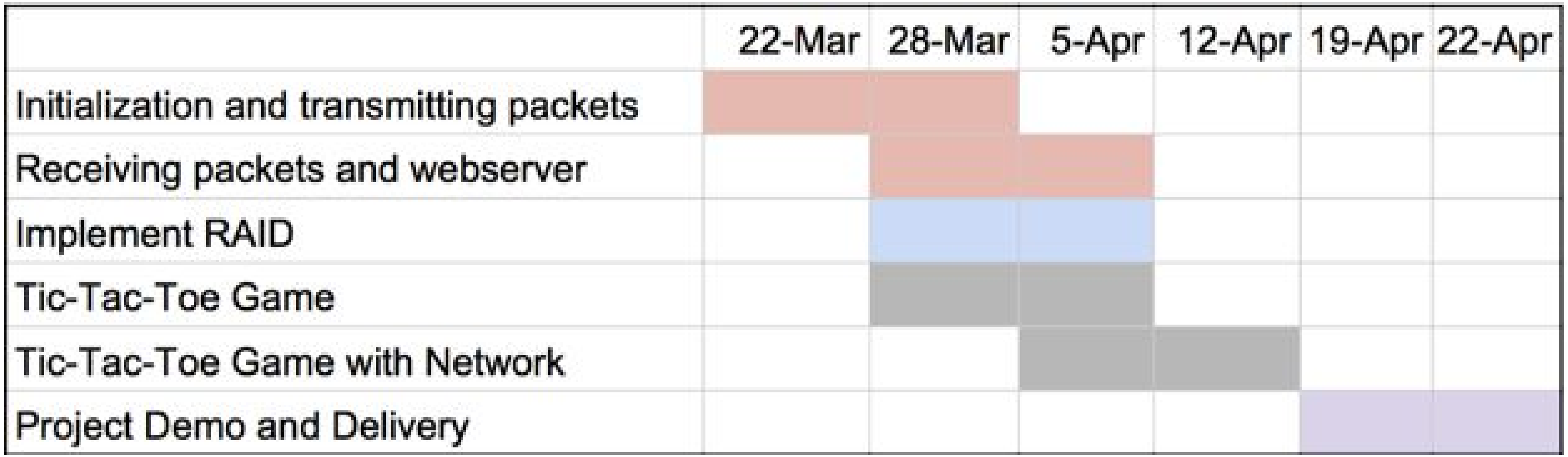
Idea

- We will implement the **Network Driver** first, so that two hosts can communicate with each other.
- Then we will implement the **RAID disk** to let the information sent by the two hosts can be safely stored on disk.
- Last we will realize the online **Tic-Tac-Toe game** so that to demonstrate how interactive gaming can be achieved in our system.

Demo Plan

1. Demonstrate the correctness of network driver by transmitting a string between two hosts.
2. Store the communicating strings on RAID disk, intentionally corrupt some bits and recover the old values.
3. Open two hosts and play Tic-Tac-Toe game to demonstrate the game logic and network ability.

Timeline



	22-Mar	28-Mar	5-Apr	12-Apr	19-Apr	22-Apr
Initialization and transmitting packets	🟥	🟥				
Receiving packets and webserver		🟥	🟥			
Implement RAID		🟦	🟦			
Tic-Tac-Toe Game		▨	▨			
Tic-Tac-Toe Game with Network			▨	▨		
Project Demo and Delivery					🟪	🟪