Speaker Notes - Design Presentation

Recap

- DaxOS is small but functional operating system and kernel that is developed for students to explore and learn systems programming.
- Our codebase is written primarily in C and x86 assembly. Additionally we also use Markdown for writing the documentation and lots of BASH shell scripts to drive our build system.
- Our compilation process uses the GCC cross-compiler which targets the i686 platform; which is a generic 32-bit P6 intel architecture.
  We also use the GNU assembler and linker from binutils package along with the ar archiver.
- Our OS is tested on qemu and developed on 64 bit linux system.
- The intended audience for our project is **students** - not actually users of the operating system.
- Our project is completed when we have implemented:
    1. VGA Display Driver with Graphics support
    2. PS/2 Keyboard Driver
    3. PS/2 Mouse Driver
    4. Limited implementation of standard C library (for eg: string.h and stdio.h)

Kernel Architecture

There are three types of kernel architectures from which we can choose:

1. Micro Kernel
2. Monolithic Kernel
3. Hybrid Kernel

The classification is based on what is in kernel space and what is in user space. There are four protection rings that represent the access a piece of software has to the hardware and system: Out of these only two rings are of interest:

1. Ring 0
   Code running in Ring 0 is said to be in supervisor mode. It has complete access to the hardware and system.
2. Ring 3
   Code running in Ring 3 is said to be in user space. It has no direct access to hardware or the system. Instead it accesses the system through system calls.

These protections are not simply implemented by the Operating system. Instead they are a part of the CPU architecture. They are activated by asm code.

In a micro kernel very limited amount of code is is in kernel space. For example, most drivers are in user space.
In a monolithic kernel device drivers and other similar modules operate in kernel space.
A hybrid kernel is somewhere in the middle of what runs in kernel space and user space.

Our OS has a monolithic design. This is motivated by two facts which are:

1. The Linux kernel is monolithic. So if we use a micro kernel architecture, students will not be able to transfer the knowledge that they acquired from our project into linux development space.
2. Our kernel operates completely in ring-0 i.e without any restrictions. Implementing user-mode is unnecessary since we have no user-mode programs. Additionally students need to be in ring-0 to understand how the hardware works.

Build Process

Our build architecture is a little involved because we are using the C programming language which does not have any real tooling for package management. Instead we have to compile and link each translation unit manually.

We use a tool called GNU Make to manage the compilation process. GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files. It does this by executing commands in a file called Makefile.

The makefile contains rules which specify what prerequisites are needed to build a particular executable and the steps required to build it. We have multiple Makefiles in our project to recursively compile and build each of our kernel components. In fact, the Makefiles by themselves account for 25% of our codebase.

The Makefiles are executed by issuing corresponding make commands (for eg: make install or make install-headers). We use shell scripts to drive the make program. We have a shell script called config.sh that sets up the system environment variables that are utilized by make.
We have other shell scripts which we'll look into in future slides.

The compilation process installs the operating system into the system root directory. This is currently the directory sysroot within our project. sysroot directory further contains two sub-directories - boot and usr. The boot directory contains DaxOS.kernel file and the usr directory contains the standard C library and unit tests. The kernel is booted by pointing the bootloader to the /boot directory.

The built kernel is tested on qemu by calling the qemu.sh shell script. This works by first packaging the kernel into a .iso image file and running qemu with the iso file.

GNU MAKE Example

The Makefile that you are seeing here generates the output DaxOS.kernel when make install-kernel command is issued in the terminal.

The make install-kernel command requires DaxOS.kernel file as a prerequisite. This means that the command can only proceed when DaxOS.kernel is available.

But we actually have a rule to produce DaxOS.kernel. This rule requires the items in variable OBJS and the file linker.ld as a prerequisite.

There is a static pattern rule that describes how files ending with .o can be produced from files ending with .c or .S. So GNU Make proceeds by first compiling the .c and .S source files to .o object files.
When this is completed we have all the prerequisites needed for producing DaxOS.kernel.

Therefore GNU Make produces DaxOS.kernel by:

1. Linking the .o files with a custom linker script (which is the linked.ld file). This script is needed because we are in a freestanding environment. In normal circumstances, this script is implicitly provided by the GCC compiler.
2. Verifying the multiboot integrity by running grub-file command.

Once DaxOS.kernel is available, all the prerequisites needed to run make install-kernel is available.
So make install-kernel proceeds by:

1. Creating /sysroot/boot directory if it does not exist.
2. Copying the DaxOS.kernel file into the directory.

Design Diagrams

UML design diagrams are best suited to describe object oriented code. Our code and language is not object oriented.
But we can still use UML concepts to describe some of the design of our project.

Use Case Diagram

This use case diagram illustrates pictorially how the user interacts with DaxOS.

The first interaction the user initiates is the boot process. This involves:

1. Setting up interrupts

The concept behind interrupts is quite simple. When a piece of hardware or software wants to interrupt the CPU to do something important, an interrupt is raised. The CPU executes a program called the ISR (Interrupt Service Routine) and returns back to what is was doing before. In order to set up interrupts the following steps have to be done at boot:

1. **Create the IDT (Interrupt Descriptor Table)**
   This table instructs the Programmable Interrupt Controller where to find the ISR for each type of interrupt. This is implemented in C as an array of structs.

2. **Remapping the PIC**
   After boot we are in protected mode. Here some of our interrupts conflict with each other. In order to prevent this we will need to remap our PIC. This is done by first sending Initialization Command Words (ICW) to the input port 0x20 and then sending the vector offsets to port 0x21.

3. **Filling the IDT**
   The IDT table is filled with the address of each ISR.

2. Loading VGA driver

There are two objectives here:

1. **Write text to the screen**
   Using VGA to print text to the screen is quite simple and is achieved by using VGA text-mode.
   We can directly write to the video memory located at address 0xB8000.
   Each character that we want to print requires a two-byte representation:

   - 1 byte, called the code-point is used to represent the character in ASCII.
   - 1 byte, is used to set the background and foreground colors. We can use 16 colors.

   In VGA text-mode we can print a maximum of 80x25 characters on the screen at one time.

2. **Drawing on the screen**
   This is accomplished by using VGA Graphics mode. Now we can plot pixels by writing to memory location 0xA0000. The maximum resolution supported is 640x480 pixels with 16 bit color support.

3. Load keyboard driver

There are two design choices for the implementation of the keyboard driver:

1. **Polling Driven**
   In Polling, the CPU keeps checking the status of the PS/2 keyboard constantly to detect a keypress.

2. **Interrupt Driven**
   When the user presses a key, the keyboard driver generates an interrupt which causes the CPU to run and ISR that handles the keypress.

Interrupt driven implementation is more efficient than polling driven keyboard drivers. Therefore we will implement an interrupt driven keyboard driver.

4. Load Mouse driver

The mouse driver is similar to the keyboard driver.

The second interaction involves the user issuing a command to the terminal. The command may require dynamic memory to be allocated.
Dynamic memory is allocated by implementing the malloc C function. The malloc function acts as a wrapper around the kernels memory manager. This needs paging and virtual memory to be enabled.

The final and last interaction between the user and the system is the shutdown sequence. This can be implemented by supporting Intel's ACPI protocol or the now depreciated APM protocol. These protocols control the amount of power each device is given. Additionally we can shutdown specific emulators by writing data to specific ports.

Activity Diagram

This diagram illustrates the shell scripts that we use to build and compile our kernel.

- The most important script here is the build.sh shell script which dives the make program to compile our source code. The build script relies further on two other shell scripts:

    1. **headers.sh script** Because we have our own version of the C standard library, we compile our kernel by instructing the GCC cross-compiler to look for system headers at our SYSROOT directory. This script compiles our custom C std-lib into an archive named libk.a and places it in the SYSROOT/usr/lib directory. Also it copies all the .h header files into SYSROOT/usr/include directory. Now we compile the kernel by passing the `--sysroot=SYSROOT` parameter.

    2. **config.sh script** This script sets up all the environment variables used by GNU Make.
    This allows us to easily change the core aspects and tooling used in our projects later. For example, we could change the compiler from GCC to CLANG by simply changing the environment variable CC to CLANG. Likewise we could also change the SYSROOT directory by changing a single variable.

    Once these two scripts are executed the build script runs the make-install command on each of the folder directory. It copies the output DaxOS.kernel file into the boot directory.

- The iso.sh script builds a bootable .iso image file from our kernel. It first calls the previously described build script and then produced an iso file by using the grub-mkrescue program. This iso file can be loaded into any emulator to run the operating system. The iso files are built into the isodir directory.

- The clean.sh script removes all the build artifacts from the compilation. Build artifacts are files that are produced by the compilation process that can always be reproduced by the compiler. Since it can be reproduced we usually remove them to maintain a clean project structure.
We have three kinds of build artifacts:

    1. **Object files** - .o files
    2. **Make dependencies** - .d files
    3. **Unwanted directories** - sysroot, isodir

    The clean script works by executing the make clean commands in each of our project directories.
    The make clean commands use the rm bash command to delete the build artifacts.

- The last script is the create-bootable-usb.sh script. This script creates a bootable USB of our operating system. It needs sudo permission and you need to pass the UNIX block file of the usb device. This of the form /dev/sda or something similar. This script formats the USB device and copies the kernel along with the GRUB bootloader into the USB device. Therefore there is a chance that you can destroy the data in the USB device if you are not careful. To prevent this we have a safety check which checks if the device has more than 10GB capacity. If this is true we do not format the drive and exit with error. If the capacity is less than 10GB, we make it bootable.

Sequence Diagram (Keyboard Driver)

This diagram illustrates the time dependent and sequential interaction between the user, the OS and the keyboard driver.

- During boot, the Operating System initializes the Keyboard driver.
  This involves:

    1. **Self Test**

        1. The keyboard device is disabled by sending command 0xAD and 0xA7 to the PS/2 controller.
        2. The PS/2 controller's output buffer is flushed by reading from port 0x60.
        3. Initiate the PS/2 controller self test by sending command 0xAA to it. A response of 0x55 indicates success.
        4. Enable the keyboard device by sending commands 0xAE and 0xA8 to the PS/2 controller.
        5. Reset the device by sending 0xFF to the keyboard.
        6. Set the LED states by sending appropriate commands.

    2. **Handling Keypress**

        When the user presses a key the keyboard device initiates an interrupt. This done by activating IRQ1 which is the standard interrupt line used by keyboards. In response to this interrupt, the CPU executes an ISR which updates a buffer with the read characters. The I/O functions in stdio.h like scanf will read from this buffer perform the necessary computation and show the results back to the user.

Sequence Diagram (User command)

After boot the user interacts with DaxOS through a command prompt like windows. It is here that the user enters commands which will be executed by the kernel. These are not processes.
We will implement the following commands:

1. **gfx_demo**
   This demo program will demonstrate some of our graphics and drawing capabilities.
2. **dcalc**
   This is a calculator application that can evaluate mathematical expressions.
3. **term_art**
   This will demonstrate some ASCII terminal art using VGA driver.

The sequence diagram illustrates the process of running the commands. All the commands use operations from standard C library. So to take input and output text to the screen the stdio functionality will be used. Then the command proceeds to do its computation. If dynamic memory is needed, the command again uses the malloc function from standard C library. The malloc function maintains a list of free stores - that is free blocks of memory. If the requested chunk of memory is already in the free store then the malloc function marks it as being used and returns a pointer to it. On the other hand if the memory in the list is not enough, the malloc function will ask the kernel to provide more memory. In this case the kernel provides a new page to the malloc function and the malloc function promptly adds the block of memory into its free store and returns a pointer to it.

Sequence Diagram (Shutdown)

Shutdown is implemented using Advanced Power Management (APM). The user initiates the shutdown command. From that point onwards there is no more interaction between the user and the OS. The OS in return asks the power management module to perform APC shutdown.

This involves calling a BIOS interrupt. In order to do that we need to first switch from protected mode to real mode. Then we need to inform the BIOS what kind of operation we want it to do. So we inform the BIOS that we want to perform APC operation by setting the register AX with value 0x5307. Next we inform the BIOS that we want the current operation to be done on all connected devices by setting register BX with value 0x0001. Lastly we put the value 0x03 in CX register to indicate shutdown operation. Now we can activate the BIOS function by raising software interrupt 0x15. Now the computer is powered off.

Modules

C has no notion of modules. But for the sake of this presentation we will consider each translation unit as a module. Some of the important modules are:

- **kernel.c** Once boot is completed, the bootloader will pass control to kernel_main function in this C file. Everything the operating system does from this point starts from this file. This file contain functions that initialize the device drivers and memory management.

- **tty.c** This is the VGA terminal driver.
  It contains:

  1. tty_initialize() function which clears all the video memory buffer.
  2. tty_put_entry_at() function which puts an ASCII character at a particular row and column.
  3. tty_write_string() function which outputs text to the screen.
  4. tty_setcolor() function to set the background and foreground color.

- **io.c** This file contains the C functions used to communicate with the I/O ports of the system. This is primarily used in the implementation of drivers.
  It contains:

  1. inportb() function which returns a byte read from the port.
  2. outportb() function which writes a byte to the specified port.

  The core of these functions are written in x86 assembly code using the **inb** and **outb** instructions. We use GCC inline assembly to write the necessary assembly code within our C function. This is more efficient than branching to another assembly function as there is no overhead associated.

- **string.h** This file contains the interface for the C library implementation of string.h. It contains the usual and familiar functions such as strncpy, strcpy, strlen and so on... The implementation of string.h is actually spread across multiple .c files. One .c file is provided for each function. For eg, strlen is implemented in strlen.c file.

Implementation plans

So now we'll talk about what we will implement by the next review.

- First thing we want to implement is a VGA text-mode driver. This will allow us to print text with colors to the screen.
  We will also support escape codes such as newline characters. Along with this we will also write the printf function in stdio library.

- We will implement the PS/2 keyboard driver. Now one problem we have is that we no longer use PS/2 devices. All HID devices that we use are actually USB devices. But we can rely on the USB Legacy Emulation feature present in most Motherboards to actually use the USB device like it was a PS/2 device. To implement the PS/2 driver we will first need to implement interrupts.

- Lastly we will also implement unit testing for our standard C library.
  Unit testing involves writing test cases for our functions. There are two challenges to implementing unit testing in our project:

  1. **Unit testing C code is pretty difficult**
     This is because there is no dependency injection in C. Dependency injection is basically eliminating the variables that an object uses and instead providing it externally.
  2. **Lack of standard C Library**
     Most unit testing frameworks in C requires the C library to work. We don't have a C library yet.

  Instead of working around these pain points we will instead come up with our own unit testing framework called d_unit. d_unit will not be dependent on the standard C library.