INTEGRATING WITH C PROGRAM:

Sections:

- KERNAL MAIN
- PUTTING IT ALL TOGETHER
- CREATING OUR OWN LIBRARIES
- PRINT FUNCTION
- ASSERTION

1)KERNAL MAIN:

we use C language to implement our kernel. C language will be the main language to use in the following lectures. We need to do some preparation first. The compiler we use is GCC and the executable file we will generate is elf file. Since the kernel assembly code is also in the kernel, we will see how to combine the assembly and C files together. The nasm assembler can output elf file. We open the build script. In the build script, we change the options to generate the elf file instead of binary file. Elf64 means we generate the elf64 object file. The name of the object file is kernel.o. Then we will link the object file with other files written in c. In the kernel assembly file, we define two sections.

```
KernelEntry:
    mov rsp,0x200000
    call KMain
```

This code we have to include in the assembly language code so that we can define main in c file.

```
section .text
extern KMain
global start
```

These are the header files we are going to include in the c program.
In the kernel we have to rearrange the code into data section and text section

Data Section:

- The data section is used to store globally defined data such as the IDT (Interrupt Descriptor Table), TSS (Task State Segment), and other data used in the kernel.
- In this step, we will copy the necessary data into the data section.

Text Section:

- The interrupt handlers, which are not defined in the kernel file, will be implemented in other files. Therefore, we remove all related code in the text section.
- This includes removing the set handler, load IDT instruction, set handler function, use entry, and the handler itself.
- Since we don't reach ring3 in the kernel entry, the code in this section is not used.
- The kernel entry is placed at the end of the code.

Initializing TSS, PIT, and PIC:

- The code proceeds to initialize the TSS, PIT (Programmable Interval Timer), and PIC (Programmable Interrupt Controller).

- These components are crucial for managing tasks, handling interrupts, and controlling hardware.

Kernel Entry and Main Function:

- After initializing the TSS, PIT, and PIC, we enter the kernel entry and jump to the kernel main function written in C.
- Before calling the main function, we adjust the kernel stack pointer to 200000.
- This new stack pointer will be used in the C code.

Exiting the Kernel:

- If the main function returns, we will stop execution at this point.
- Normally, the main function in the kernel will not return, and the system will continue running indefinitely.

By organizing the code into separate sections, initializing the necessary components, and jumping to the main function, we ensure that the kernel is prepared to execute the desired operations and handle interrupts and tasks effectively.

```
OUTPUT_FORMAT("elf64-x86-64")
ENTRY(start)

SECTIONS
{
    . = 0x200000
    .text : {
        *(.text)
    }

    .rodata : {
        *(.rodata)
    }

    . = ALIGN(16);
    .data : {
        *(.data)
    }

    .bss : {
        *(.bss)
    }
}
```

Linker file code that should be included to show the output.

Linker Script: linker.lds

- The linker script, named linker.lds, is created to define the layout and organization of the output file.
- The first step is to specify the output format by writing "output format elf64-x86-64".
- Next, we indicate that the label "start" is the entry point of the kernel by typing "entry start".
- We utilize the linker script to set the desired address for the kernel. Although the kernel assembly file does not use the "org" directive to specify the address 200000, the loader file still jumps to that address after loading the kernel. Therefore, we use the linker script to achieve this.
- Multiple sections are defined in the files, and we specify them in the linker script using the "sections" keyword.

- The first segment we define is the text segment, which contains the ".text" sections from the object files (kernel.o and main.o). We use the syntax "*(.text)" to add all the ".text" sections to the text segment.
- To set the code running at address 200000, we use the expression ". = 200000" to set the current location to 200000.
- The next segment is the read-only data segment, where we include all the ".rodata" sections.
- We define the data segment to include the necessary data sections.
- Lastly, we specify the BSS (Block Started by Symbol) segment.
- Additionally, we ensure that the data section is aligned to a 16-byte boundary by writing ". = align 16".
- This concludes the creation of the linker script.

Build Script:

- In the build script, we add the option "-T" followed by the name of the linker script we just created.
- This instructs the build process to use the specified linker script for linking the object files and generating the final output file.
-

2)PUTTING TOGETHER IN C:

System V AMD64 Calling Convention:

- The System V AMD64 calling convention is used for function calls in the AMD64 architecture.
- The first six arguments of a function are stored in registers: rdi, rsi, rdx, rcx, r8, and r9.
- Additional arguments beyond the first six are passed on the stack in reverse order.
- The return value of a function is stored in the rax register.
- Certain registers, including rax, rcx, rdx, rsi, and others, are known as caller-saved registers. This means that the caller is responsible for saving the values of these registers if it calls other functions that may modify them. Typically, the caller saves the values on the stack before making the function call and restores them after the function returns.
- Other registers, such as rbx, rbp, r12, and others, are called callee-saved registers. The values of these registers are preserved across function calls, meaning that a function called from another function is responsible for preserving the values of these registers if it modifies them.

```
#ifndef _TRAP_H_
#define _TRAP_H_

#include "stdint.h"
```

Header File trap.h:

- The header file "trap.h" is created to define interrupt structures that are needed when dealing with interrupts in C files.
- The header file starts with an include guard to ensure that it is included only once. This guard prevents multiple definitions if the file is included multiple times.
- The trap.h file defines interrupt structures that can be used in C files.
- As the interrupt vector numbers were removed from the original file, trap.h will now include these vector numbers.
- Each vector number corresponds to a specific interrupt or exception, such as vector 0 representing the zero exception.
- In trap.h, the defined vector numbers are pushed onto the stack, along with pushing the general-purpose registers.
- The code also includes an "end of interrupt" command that sends the EOI (End of Interrupt) signal to the PIC (Programmable Interrupt Controller).
- Reading the ISR (Interrupt Service Routine) is necessary for processing spurious interrupts, so the ISR is read in this file.
- The last procedure defined in trap.h is "load idt." This procedure is required because loading the IDT (Interrupt Descriptor Table) directly in C using the "load idt" instruction is not possible.
- Instead, a separate procedure called "load idt" is defined in trap.h, which can be called from a C file to load the IDT. This procedure takes the address of the IDT as a parameter.

3)DEFINING OUR OWN LIBRARIES:

We are going to define some of the function such as

- memcmp
- memset
- memmove
- memcopy.

MEMSET:

```
memset:                rdi (buffer)  rsi (value)   rdx (size)
    cld
    mov ecx,edx
    mov al,sil
    rep
```

- The memset function fills a block of memory with a specified value. The first parameter of the function represents the destination where the memory will be filled. The second parameter specifies the value to be set. The third parameter indicates the size of the memory block in bytes.

MEMCPM:

```
memcmp:                rdi (src1)  rsi (src2)   rdx (size)
    cld
    xor eax,eax
    mov ecx,edx
    repe
```

- The memcmp function compares two blocks of memory. It returns an integer value as the result of the comparison. If the return value is 0, it indicates that the compared blocks are equal. Otherwise, they are considered not equal. The first two parameters of the function

represent the addresses of the two memory areas being compared, and the size of the memory blocks is specified in the last parameter.

MEMMOVE:

```
memmove:
    cld
    cmp rsi,rdi
    jae .copy
    mov r8,rsi
    add r8,rdx
    cmp r8,rdi
    jbe .copy

.overlap:
    std
    add rdi,rdx
    add rsi,rdx
    sub rdi,1
    sub rsi,1

.copy:
    mov ecx,edx
    rep movsb
    cld
    ret
```

- The memcpy function copies a block of memory from a source location to a destination location. The first parameter of the function represents the destination where the data will be copied, and the second parameter represents the source from where the data will be read. The third parameter specifies the size of the data to be copied in bytes.

These functions provide basic memory manipulation operations commonly used in programming to copy, fill, and compare memory blocks efficiently.

4)PRINT FUNCTION:

Structure of header file for print

```
#ifndef _PRINT_H_
#define _PRINT_H_

#define LINE_SIZE 160

struct ScreenBuffer {
    char* buffer;
    int column;
    int row;
};

int printk(const char *format, ...);

#endif
```

The header file "print.h" is created with the following contents:

- Guard: A guard is added to ensure that the header file is included only once.
- Constant: The constant "line_size" is defined as 160. This constant represents the number of bytes in a line in text mode. In text mode, there are 80 characters per line, and each character takes up 2 bytes.
- Structure: The structure "screen_buffer" is defined to handle printing messages on the screen. It contains three fields:
- "buffer": Represents the address of the screen buffer, which is set to "b8000" in this case.
- "column": Represents the current column position where the next message will be printed.
- "row": Represents the current row position where the next message will be printed.

- Function: The "printk" function is introduced as the system's printing function used in kernel mode. It is a variable argument function, indicated by the use of "..." in its parameter list. The first parameter is the address of the format string, which is declared as "const" to indicate that the data pointed to by the format should not be modified. The function uses the format string to interpret the variable arguments provided. It returns the count of the characters it actually prints.

By including this header file, the system can utilize the "printk" function and the "screen_buffer" structure to print messages on the screen in kernel mode.

C FILE:

```
}
else {
    switch (format[++i]) {
        case 'x':
            integer = va_arg(args, int64_t);
            buffer_size += hex_to_string(buffer, buffer_size, (uint64_t)integer);
            break;

        case 'u':
            integer = va_arg(args, int64_t);
            buffer_size += udecimal_to_string(buffer, buffer_size, (uint64_t)integer);
            break;

        case 'd':
            integer = va_arg(args, int64_t);
            buffer_size += decimal_to_string(buffer, buffer_size, integer);
            break;

        case 's':
            string = va_arg(args, char*);
            buffer_size += read_string(buffer, buffer_size, string);
            break;

        default:
            buffer[buffer_size++] = '%';
```

In the "printk" function, when processing the format string, the following steps are taken:

1. Copying Characters: Each character in the format string is examined. If it is not the '%' character, it is simply copied to the buffer, the buffer size is incremented, and the next character is processed.

2. Specifier Handling: When encountering the '%' character, the next character is checked to determine the specifier type. This is done using a switch statement. Supported specifiers include:

- 'x': Hexadecimal type
- 'u': Unsigned integer type
- 'd': Signed integer type
- 's': String specifier

3. Default Case: If the specifier is not one of the supported types, the default case is executed. Here, the '%' character is copied to the buffer, the index is decremented, and the loop continues to process the next character.

4. String Specifier Handling: When the specifier is 's', the corresponding argument is retrieved from the variable arguments list using the `va_arg` macro. The first argument to `va_arg` is the `args` variable, and the second argument specifies the type of the variable to retrieve, which is a pointer to type `char`. The characters from the retrieved pointer are then read and copied to the buffer.

By following these steps, the "printk" function can process the format string, handle different specifiers, and copy the appropriate data to the buffer for printing.

```c
static int hex_to_string(char *buffer, int position, uint64_t digits)
{
    char digits_buffer[25];
    char digits_map[16] = "0123456789ABCDEF";
    int size = 0;

    do {
        digits_buffer[size++] = digits_map[digits % 16];
        digits /= 16;
    } while (digits != 0);

    for (int i = size-1; i >= 0; i--) {
        buffer[position++] = digits_buffer[i];
    }

    buffer[position++] = 'H';

    return size+1;
}
```

```c
static int udecimal_to_string(char *buffer, int position, uint64_t digits)
{
    char digits_map[10] = "0123456789";
    char digits_buffer[25];
    int size = 0;

    do {
        digits_buffer[size++] = digits_map[digits % 10];
        digits /= 10;
    } while (digits != 0);

    for (int i = size-1; i >= 0; i--) {
        buffer[position++] = digits_buffer[i];
    }

    return size;
}
```

In the "decimalToString" function, the following steps are taken to convert a signed integer to a string:

1. Digit Mapping: A digit map is defined to map the decimal digits (0-9) to their corresponding characters. This mapping is used to convert the digits to characters later in the process.

2. Conversion Loop: The function uses a do-while loop to convert the decimal value. Instead of taking the modulus by 16 (as in

hexadecimal conversion), the modulus by 10 is used to obtain the digits. The corresponding character is then copied to the buffer.

3. Digit Update: After obtaining the least significant digit, the value is divided by 10, effectively shifting the digits to the right. This process is repeated as long as the value is non-zero, allowing all the digits to be obtained.

4. Reversing the Order: The characters obtained from the conversion loop are in reverse order, starting from the least significant digit. To correct the order, a for loop is used to copy the characters to the buffer from the last character to the first one.

5. Returning the Size: Finally, the function returns the size, which represents the number of characters that were copied to the buffer.

This "decimalToString" function is used to convert signed integers to their string representation, complementing the "unsignedDecimalToString" function for handling unsigned integers.

```
static int decimal_to_string(char *buffer, int position, int64_t digits)
{
    int size = 0;

    if (digits < 0) {
        digits = -digits;
        buffer[position++] = '-';
        size = 1;
    }

    size += udecimal_to_string(buffer, position, (uint64_t)digits);
    return size;
}
```

In the function "decimalToString," the following steps are taken to convert a signed integer to a string:

1. Checking the Sign: The function first checks if the value passed in is positive or negative. If it is negative, the value is converted to its positive form. Additionally, a minus sign is added as the first character to indicate the negative value. The minus sign is copied to the buffer, and the size is updated to 1.

2. Calling Unsigned Decimal to String: After handling the sign, the actual conversion of the positive value is done by calling the "unsignedDecimalToString" function. The buffer, position, and the digits to convert are passed as parameters. This function handles the conversion of unsigned integers and returns the number of characters it actually prints.

3. Updating the Size and Returning: If the value passed in was positive, the size is updated with the return value of the "unsignedDecimalToString" function (representing the number of characters printed). Finally, the updated size is returned.

By checking the sign and then calling the unsigned conversion function, the "decimalToString" function effectively converts both positive and negative signed integers to their string representation.

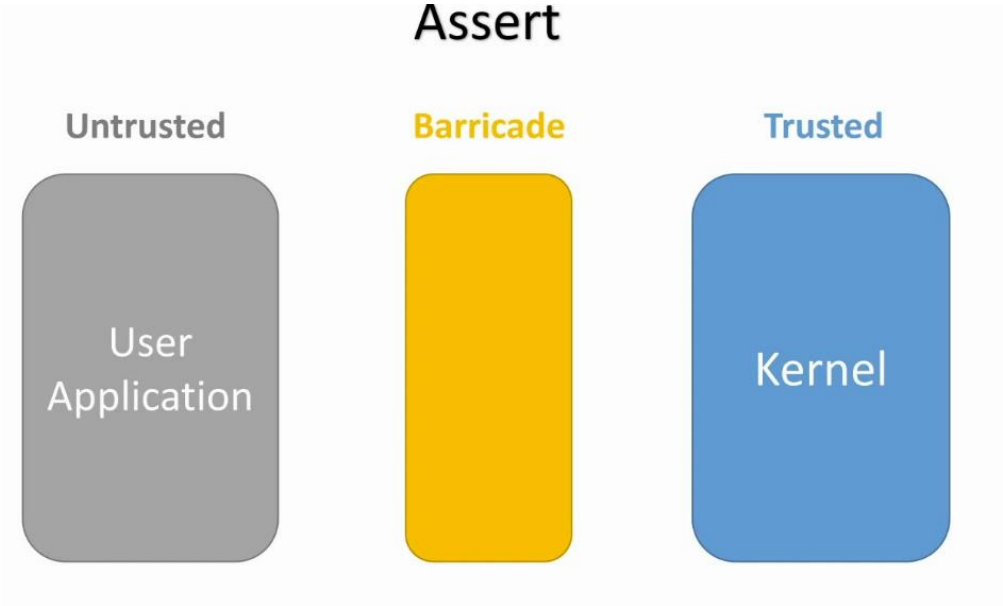
```
static void write_screen(const char *buffer, int size, struct ScreenBuffer *sb, char color)
{
    int column = sb->column;
    int row = sb->row;

    for (int i = 0; i < size; i++) {
        if (row >= 25) {
            memcpy(sb->buffer,sb->buffer+LINE_SIZE,LINE_SIZE*24);
            memset(sb->buffer+LINE_SIZE*24,0,LINE_SIZE);
            row--;
        }

        if (buffer[i] == '\n') {
            column = 0;
            row++;
        }
```

This function we used for writing the characters on to the screen. These are the definitions for all the function. That are mentioned int the main function.

5)ASSERTIONS:



1. Importance of Assertions: Assertions are used throughout the modules of the kernel to ensure that everything runs as expected. In a complex system like a kernel, even small errors can have significant consequences. Assertions help identify and catch these errors, preventing the system from running with faulty assumptions.

2. System Structure: The system consists of user programs in the untrusted zone and the kernel in the trusted area. User programs make requests to the kernel via system calls, and the barricade checks the validity of these requests and associated data. If the checks fail, an error message is returned to the programs. Valid requests are processed by the kernel.

3. Purpose of Assertions: Assertions are used within the trusted area of the kernel. Unlike the checks performed by the barricade, assertions focus on identifying errors within the kernel itself. When an assertion fails, it stops the system and prints the file name and line number where the error occurred. This information helps in locating and resolving the error during the development process.

4. Implementation of Assert: The assert function is implemented as a macro. It is defined in the "debug.h" header file and takes an expression as its argument. The macro evaluates the expression and if it evaluates to false (0), the assertion fails and triggers an error, stopping the system and displaying the file name and line number associated with the failed assertion.

By using assertions, the kernel developers can detect and fix errors during the development phase, ensuring a more robust and reliable system.

```
#ifndef _DEBUG_H_
#define _DEBUG_H_

#include "stdint.h"

#define ASSERT(e) do {                      \
        if (!(e))                           \
            error_check(__FILE__,__LINE__); \
} while (0)

void error_check(char *file, uint64_t line);

#endif
```

Predefined macro (filename and the line number)

For assertions we use header file debug.h

Structure of the debug.h

In c file we will include header file debug.h.

In the assert function, there are two main actions performed:

1. Print Error Message: The function begins by printing an error message to provide information about the failed assertion. This can include additional details to make it more readable and informative for debugging purposes. For example, it may include a message such as "Assertion failed!" or "Error check failed!".

2. Jump to Infinite Loop: After printing the error message and any associated details, the function jumps to an infinite loop. This effectively halts the system's execution at that point, preventing further code execution. The purpose of this infinite loop is to stop the system and allow developers to investigate and address the error.

As part of the error message, the function also includes the name of the file and the line number where the assertion failed. This information helps in identifying the specific location in the code where the assertion condition was not met, aiding in the debugging process.