Writing the Process Code for Operating System

The process code for an operating system is a crucial component responsible for managing various processes running on the system. Writing efficient and optimized process code is essential for ensuring the smooth functioning of the operating system.

Before writing the process code, one needs to have a clear understanding of the system's architecture and the requirements of the processes that need to be managed. It is crucial to identify the various resources required by the processes and allocate them in a manner that maximizes their utilization.

One of the key aspects of writing efficient process code is minimizing the time and resources required to switch between processes. This can be achieved through the use of appropriate data structures and algorithms, such as queueing and scheduling mechanisms.

Another important consideration is ensuring the security and stability of the operating system. The process code must be designed to prevent unauthorized access to critical system resources and to handle errors and exceptions gracefully.

To ensure the quality and reliability of the process code, it is essential to follow best practices and industry standards. This includes writing clear and concise code, incorporating error handling and testing procedures, and adhering to established coding conventions and guidelines.

In conclusion, writing efficient and optimized process code is a critical component of operating system development. By following best practices and utilizing appropriate data structures and algorithms, developers can ensure the stability, security, and efficiency of the operating system.

boot.asm

Let's go through each section and explain its purpose:

```
[BITS 16]
[ORG 0x7c00]
```

These directives set the assembler to operate in 16-bit mode and specify the origin of the program at memory address 0x7c00. This is the conventional location where a boot sector is loaded by the BIOS during system startup.

```
start:
    xor ax, ax
    mov ds, ax
    mov es, ax
    mov ss, ax
    mov sp, 0x7c00
```

The `start` label marks the beginning of the code. These instructions initialize the data segment (DS), extra segment (ES), and stack segment (SS) registers to zero, and set the stack pointer (SP) to 0x7c00, which is the address of the boot sector in memory.

```
TestDiskExtension:
    mov [DriveId], dl
    mov ah, 0x41
    mov bx, 0x55aa
    int 0x13
    jc NotSupport
    cmp bx, 0xaa55
    jne NotSupport
```

This section tests the disk extension functionality using the BIOS interrupt 0x13, which provides disk services. It saves the drive number in the `DriveId` variable, sets the AH register to 0x41 (check extensions), and performs the interrupt. If the carry flag is set (indicated by the `jc` instruction), it jumps to the `NotSupport` label. Otherwise, it compares the value in the BX register to 0xaa55, and if they are not equal, it also jumps to `NotSupport`.

```
LoadLoader:
    mov si, ReadPacket
    mov word [si], 0x10
    mov word [si+2], 5
    mov word [si+4], 0x7e00
    mov word [si+6], 0
    mov dword [si+8], 1
    mov dword [si+0xc], 0
    mov dl, [DriveId]
    mov ah, 0x42
    int 0x13
```

```
    jc ReadError

    mov dl, [DriveId]
    jmp 0x7e00
```

This section loads the next stage of the bootloader from the disk. It sets up a packet in memory at the `ReadPacket` label, specifying the number of sectors to read, the destination memory address (0x7e00), and the drive number. It then calls interrupt 0x13 with AH=0x42 (extended read) to read the sectors. If the carry flag is set, it jumps to `ReadError`. Otherwise, it reloads the drive number into DL and jumps to the address 0x7e00, where the next stage is assumed to be loaded.

```
ReadError:
NotSupport:
    mov ah, 0x13
    mov al, 1
    mov bx, 0xa
    xor dx, dx
    mov bp, Message
    mov cx, MessageLen
    int 0x10
```

These labels mark the locations where the program branches in case of a read error or unsupported disk extension. In either case, the code displays an error message on the screen using BIOS interrupt 0x10 (video services).

```
End:
    hlt
    jmp End
```

The `End` label is the end of the program. It halts the processor using the `hlt` instruction and

jumps back to itself, creating an infinite loop to prevent the CPU from executing any random code beyond the bootloader.

```
DriveId:    db 0
Message:    db "We have an error in boot process"
MessageLen: equ $-Message
ReadPacket: times 16 db 0
```

These are data declarations. `DriveId` is a single byte reserved for storing the drive number. `Message` is a string that holds the error message. `MessageLen` is a symbol defined as the length of the `Message` string. `ReadPacket` is a buffer of 16 bytes initialized with zeros.

The final section of the code consists of various declarations that fill the remaining space of the boot sector with zeros, followed by specific data structures that are used by the bootloader or by the subsequent stages.

kernel.asm

Let's go through each section and explain its purpose:

```
section .data
global Tss

Gdt64:
    dq 0
    dq 0x0020980000000000
    dq 0x0020f80000000000
    dq 0x0000f20000000000
TssDesc:
    dw TssLen-1
    dw 0
    db 0
    db 0x89
    db 0
    db 0
    dq 0

Gdt64Len: equ $-Gdt64

Gdt64Ptr: dw Gdt64Len-1
          dq Gdt64

Tss:
    dd 0
    dq 0xffff800000190000
    times 88 db 0
    dd TssLen

TssLen: equ $-Tss
```

In the `.data` section, the code defines and initializes the Global Descriptor Table (GDT) entries and the Task State Segment (TSS) descriptor. The GDT entries are stored in the

`Gdt64` array, and their lengths are calculated using the `equ` directive. The `Gdt64Ptr` holds the length and base address of the GDT.

The TSS is defined in the `Tss` structure, which includes the initial values for various TSS fields. The `TssLen` symbol calculates the length of the TSS structure.

```
section .text
extern KMain
global start

start:
    mov rax, Gdt64Ptr
    lgdt [rax]
```

In the `.text` section, the `start` label marks the beginning of the code. The `lgdt` instruction loads the GDT pointer from the memory location specified by `Gdt64Ptr` into the GDTR register, which sets up the GDT for the kernel.

```
SetTss:
    mov rax, Tss
    mov rdi, TssDesc
    mov [rdi+2], ax
    shr rax, 16
    mov [rdi+4], al
    shr rax, 8
    mov [rdi+7], al
    shr rax, 8
    mov [rdi+8], eax
    mov ax, 0x20
    ltr ax
```

The `SetTss` routine sets up the TSS descriptor. It moves the base address of the TSS to `rax` and sets up the TSS descriptor in the `TssDesc` structure by storing the appropriate values at specific offsets.

The `ltr` instruction loads the task register (TR) with the selector (`ax`) to make the processor aware of the TSS.

```
InitPIT:
    mov al, (1<<2) | (3<<4)
    out 0x43, al

    mov ax, 11931
```

```
    out 0x40, al
    mov al, ah
    out 0x40, al
```

The `InitPIT` routine initializes the Programmable Interval Timer (PIT). It configures the PIT by writing appropriate values to the control port (`0x43`) and data ports (`0x40`).

```
InitPIC:
    mov al, 0x11
    out 0x20, al
    out 0xa0, al

    mov al, 32
    out 0x21, al
    mov al, 40
    out

0xa1, al

    mov al, 4
    out 0x21, al
    mov al, 2
    out 0xa1, al

    mov al, 1
    out 0x21, al
    out 0xa1, al

    mov al, 0b11111110
    out 0x21, al
    mov al, 0b11111111
    out 0xa1, al
```

The `InitPIC` routine initializes the Programmable Interrupt Controllers (PIC). It sends specific initialization codes to both the master PIC (ports `0x20` and `0x21`) and the slave PIC (ports `0xa0` and `0xa1`) to set up interrupt handling.

```
mov rax, KernelEntry
push 8
push rax
db 0x48
retf

KernelEntry:
    mov rsp, 0xffff800000200000
    call KMain
```

The code sets up the kernel entry point at `KernelEntry`. It pushes the appropriate values onto the stack, including the code segment selector (`8`) and the address of `KernelEntry` itself. The `retf` instruction performs a far return, which transfers control to the specified segment and offset.

Inside `KernelEntry`, the code sets the stack pointer (`rsp`) to a specific memory address and calls the `KMain` function.

```
End:
    hlt
    jmp End
```

The `End` label represents the end of the program. The `hlt` instruction halts the CPU, and the `jmp End` instruction creates an infinite loop to prevent the CPU from executing any random code beyond the kernel.

This code snippet sets up various system structures, initializes hardware components (PIT and PIC), and calls the `KMain` function, which is likely the entry point of the kernel.

lib.asm

The provided code snippet implements several memory-related functions: `memset`, `memcpy`, `memmove`, and `memcmp`. Let's explain each function:

```
memset:
    cld
    mov ecx, edx
    mov al, sil
    rep stosb
    ret
```

The `memset` function is used to set a block of memory to a specified value. It uses the `stosb` instruction to store the value in the `al` register (`sil`) into the memory block pointed to by `edx`. The `rep` prefix with `stosb` repeats the store operation `ecx` times. The function then returns.

```
memcmp:
    cld
    xor eax, eax
```

```
    mov ecx, edx
    repe cmpsb
    setnz al
    ret
```

The `memcmp` function compares two memory blocks pointed to by `esi` and `edi` with a size specified by `edx`. It uses the `cmpsb` instruction to compare bytes in the memory blocks. The `repe` prefix repeats the comparison until the end of the memory blocks or until a mismatch is found. The `setnz al` instruction sets the `al` register to 1 if a mismatch is found, or 0 otherwise. The function then returns.

```
memcpy:
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
```

Both `memcpy` and `memmove` functions perform memory copying. The code first compares the source address (`rsi`) with the destination address (`rdi`). If the destination address is greater than or equal to the source address (`jbe .copy`), it jumps directly to the copy operation.

If there is an overlap (source and destination addresses overlap), the code handles it by adjusting the source and destination addresses using pointer arithmetic. It goes into the `.overlap` section and sets the direction flag (`std`) to copy the memory block backward.

Finally, in the `.copy` section, it uses the `movsb` instruction to copy bytes from the source address to the destination address, repeating the operation `ecx` times. The `cld` instruction clears the direction flag to ensure normal string operations. The function then returns.

These functions provide basic memory manipulation operations commonly used in programming. They are implemented using string instructions to optimize performance by utilizing repetitive memory operations.

loader.asm

Let's go through the code step by step:

```
[BITS 16]
[ORG 0x7e00]
```

This specifies that the code is written for 16-bit real mode and sets the origin (start address) of the code to 0x7e00.

```
start:
    mov [DriveId], dl
```

This instruction moves the value of the DL register (drive number) into the memory location pointed to by the DriveId label. It saves the drive number for later use.

```
mov eax, 0x80000000
cpuid
cmp eax, 0x80000001
jb NotSupport
```

These instructions use the CPUID instruction to check for CPU feature support. The CPUID instruction returns information about the CPU capabilities. The code checks if the CPU supports extended function calls by checking the value of the EAX register after executing the CPUID instruction. If the value is less than 0x80000001, it jumps to the NotSupport label.

```
mov eax, 0x80000001
cpuid
test edx, (1<<29)
jz NotSupport
test edx, (1<<26)
jz NotSupport
```

These instructions use the CPUID instruction again to check for specific CPU features. It checks if the CPU supports the "Long Mode" (bit 29 of the EDX register) and "NX bit" (bit 26 of the EDX register). If either of these features is not supported, it jumps to the NotSupport label.

```
LoadKernel:
    ; ... code omitted ...
```

This section of the code is responsible for loading the kernel. It sets up a read packet with the necessary parameters (such as buffer address, sector count, etc.) and invokes the BIOS interrupt 0x13 with the AH register set to 0x42. This interrupt call is used to read data from the disk. If the read operation fails (indicated by the carry flag being set), it jumps to the ReadError label.

```
LoadUser:
    ; ... code omitted ...
```

Similar to LoadKernel, this section loads the user program from the disk using the read packet and BIOS interrupt 0x13.

```
GetMemInfoStart:
    ; ... code omitted ...
```

This section retrieves memory information using the BIOS interrupt 0x15 and the E820 memory map. It uses the GetMemInfoStart label to start the loop and the GetMemInfo label to iterate and retrieve memory information. The retrieved memory information is stored in memory starting at the address 0x9000.

```
TestA20:
    ; ... code omitted ...
```

This section checks and enables the A20 line. It verifies if the A20 line is already enabled by checking memory values. If the A20 line is not enabled, it sets the A20 line and jumps to the End label.

```
SetVideoMode:
    mov ax, 3
    int 0x10
```

This sets the video mode to mode 3 (text mode) using BIOS interrupt 0x10. It switches the video mode to prepare for displaying text on the screen.

```
cli
lgdt [Gdt32Ptr]
lidt [Idt32Ptr]

mov eax, cr0
or eax, 1
mov cr0, eax
```

These instructions perform some basic initialization before transitioning to protected mode. They clear the interrupt flag (CLI), load the Global Descriptor Table (GDT) and Interrupt Descriptor Table (IDT) using the LGDT and LIDT instructions respectively.

Then, they modify the Control Register 0 (CR0) to enable protected mode by setting the first bit (PG bit) to 1.

```
jmp 8:PMEntry
```

This jumps to the PMEntry label in 32-bit protected mode.

The code continues with 32-bit protected mode instructions and eventually transitions to 64-bit long mode with the LMEntry label.

Please note that the provided code is a bootloader and assumes certain hardware and firmware behavior. It loads a kernel and user program, performs memory checks, and

transitions to protected mode and long mode. It is designed to run as a standalone bootloader and execute the loaded code.

trap.asm

The code you provided is an interrupt handler routine written in assembly language. It defines several interrupt vectors and their corresponding handlers. Let's go through the code:

```
section .text
extern handler
```

This code section defines the ".text" section and declares an external symbol called "handler". The "handler" symbol is assumed to be defined elsewhere.

```
global vector0
global vector1
global vector2
global vector3
global vector4
global vector5
global vector6
global vector7
global vector8
global vector10
global vector11
global vector12
global vector13
global vector14
global vector16
global vector17
global vector18
global vector19
global vector32
global vector39
global sysint
global eoi
global read_isr
global load_idt
global load_cr3
global pstart
global read_cr2
```

These lines declare global symbols that represent the various interrupt vectors and functions used in the code.

```
Trap:
    ; Save registers on the stack
    push rax
    push rbx
    ; ... other registers omitted ...
    push r15

    ; Save the stack pointer in rdi and call the handler function
    mov rdi, rsp
    call handler

TrapReturn:
    ; Restore registers from the stack
    pop r15
    ; ... other registers omitted ...
    pop rbx
    pop rax

    ; Adjust the stack and return from the interrupt
    add rsp, 16
    iretq
```

The "Trap" routine is the common entry point for all the interrupt vectors. It saves registers on the stack and then calls the "handler" function, passing the stack pointer as an argument in the rdi register. After the handler returns, it restores the registers from the stack and adjusts the stack pointer before returning from the interrupt using the "iretq" instruction.

```
vector0:
    push 0
    push 0
    jmp Trap

vector1:
    ; ... other vector handlers omitted ...
```

These lines define the various interrupt vectors. Each vector pushes specific values onto the stack (e.g., error code, interrupt number) and then jumps to the common "Trap" routine. The values pushed onto the stack can be accessed by the "handler" function to handle the specific interrupt.

```
eoi:
    mov al, 0x20
    out 0x20, al
    ret
```

The "eoi" function is used to send an End-of-Interrupt (EOI) signal to the Programmable Interrupt Controller (PIC). It writes a value of 0x20 to the PIC's command port (0x20) to acknowledge the interrupt and then returns.

```
load_idt:
    lidt [rdi]
    ret
```

The "load_idt" function loads the Interrupt Descriptor Table (IDT) by using the lidt instruction. It expects the address of the IDT descriptor to be passed in the rdi register.

```
load_cr3:
    mov rax, rdi
    mov cr3, rax
    ret
```

The "load_cr3" function loads the value passed in the rdi register into the Control Register 3 (CR3), which is responsible for holding the physical address of the page directory table used for address translation. It then returns.

```
read_cr2:
    mov rax, cr2
    ret
```

The "read_cr2" function reads the value of Control Register 2 (CR2), which holds the address causing a page fault, and returns the value in

the rax register.

```
pstart:
    mov rsp, rdi
    jmp TrapReturn
```

The "pstart" function sets the stack pointer (rsp) to the value passed in the rdi register and then jumps to the "TrapReturn" label, effectively returning from the interrupt.

Overall, this code provides a framework for handling various interrupts and system calls. It saves the necessary register state, calls the "handler" function, and restores the register state before returning from the interrupt. Additionally, it includes functions to handle interrupt acknowledgment, loading the IDT and CR3, and reading the CR2 register.

debug files

Let's go through the code in more detail:

In "debug.h":

```
#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
```

- The `#ifndef` and `#define` directives are used to prevent multiple inclusions of the same header file.
- The `stdint.h` header file is included to provide the definition of the `uint64_t` type.
- The `ASSERT(e)` macro is defined. It takes a condition `e` as its argument.
- Inside the macro, the condition `!(e)` is checked. If the condition is false (evaluates to 0), indicating an assertion failure, the `error_check()` function is called with the current file name (`__FILE__`) and line number (`__LINE__`).
- The macro is wrapped in a do-while(0) loop to ensure proper semicolon handling.

The "error_check" function:

```
void error_check(char *file, uint64_t line)
{
    printk("\\n----------------------------------------\\n");
    printk("             ERROR CHECK");
    printk("\\n----------------------------------------\\n");
    printk("Assertion Failed [%s:%u]", file, line);

    while (1) { }
}
```

- The `error_check` function takes two parameters: a pointer to a char array representing the file name and a 64-bit unsigned integer representing the line number.
- Inside the function, an error message is printed using the `printk()` function. The message includes a header indicating "ERROR CHECK", followed by the file name and line number where the assertion failed.
- Finally, an infinite loop (`while (1)`) is entered to halt the program execution indefinitely.

In summary, this code provides a simple debugging mechanism through the `ASSERT` macro. When an assertion fails (the condition provided to `ASSERT` evaluates to false), the `error_check` function is called. This function prints an error message indicating the assertion failure and halts the program. This can be helpful during debugging to identify and diagnose errors by providing a clear indication of where an assertion failed in the code.

memory module

Let's break down the code in detail:

In "memory.h":

- The header files `stdint.h`, `stddef.h`, and `stdbool.h` are included to provide necessary type definitions and boolean values.
- Four structures are defined:
    - `struct E820`: Represents a memory region with `address`, `length`, and `type` fields.

    - `struct FreeMemRegion`: Represents a free memory region with `address` and `length` fields.
    - `struct Page`: Represents a page in memory with a `next` pointer.
- Three type definitions are provided:
    - `PDE`: Represents a Page Directory Entry.
    - `PD`: Represents a Page Directory.
    - `PDPTR`: Represents a Page Directory Pointer Table.
- Several macros are defined:
    - `PTE_P`, `PTE_W`, `PTE_U`: Constants for page table entry flags.
    - `PTE_ENTRY`: A constant indicating a page table entry is present.
    - `KERNEL_BASE`: The base address of the kernel.
    - `PAGE_SIZE`: The size of a page in bytes.
    - `PA_UP(v)`: Rounds up the virtual address `v` to the next page boundary.
    - `PA_DOWN(v)`: Rounds down the virtual address `v` to the previous page boundary.
    - `P2V(p)`: Converts a physical address `p` to a virtual address in kernel space.
    - `V2P(v)`: Converts a virtual address `v` in kernel space to a physical address.
    - `PDE_ADDR(p)`: Extracts the physical address of a Page Directory Entry `p`.
    - `PTE_ADDR(p)`: Extracts the physical address of a Page Table Entry `p`.
- Function declarations for memory management functions.

In "memory.c":

- Static function `free_region(uint64_t v, uint64_t e)`: Frees memory pages in the region from `v` to `e`.
- Global variables:
    - `free_mem_region`: An array of `struct FreeMemRegion` representing free memory regions.
    - `free_memory`: A `struct Page` representing the first free memory page.

    - `memory_end`: The end address of the memory.
- Function `init_memory()`: Initializes the memory by populating `free_mem_region` and freeing memory pages.
- Function `kfree(uint64_t v)`: Frees a memory page at virtual address `v` by adding it to the list of free memory pages.
- Function `kalloc()`: Allocates a memory page from the list of free memory pages.
- Functions `find_pml4t_entry()`, `find_pdpt_entry()`, and `map_pages()`: Helper functions for mapping virtual addresses to physical addresses in the page tables.
- Function `switch_vm(uint64_t map)`: Switches the current virtual memory context to the one specified by `map`.
- Function `setup_kvm()`: Sets up the kernel's virtual memory mapping by allocating a new page table and mapping the kernel's memory.
- Function `init_kvm()`: Initializes the kernel's virtual memory by calling `setup_kvm()` and switching to the new page table.
- Function `setup_uvm(uint64_t map, uint64_t start, int size)`: Sets up a user's virtual memory mapping by allocating a new page table, mapping the user's memory, and copying the content to the new mapping.
- Functions `free_pages()`, `free_pdt()`, `free_pdpt()`, and `free_pml4t()`: Helper functions for freeing memory pages and page tables.
- Function `free_vm(uint64_t map)`: Frees the virtual memory context specified by `map` by freeing memory pages and page tables.

The code provides functions for initializing and managing memory, allocating and freeing memory pages, mapping virtual addresses to physical addresses, and switching between different virtual memory contexts. It also includes helper functions for working with page tables and page table entries.

print functionalities:

Here's a breakdown of the code:

In "print.h":

- The header file includes the necessary dependencies: `stdint.h`, `stdarg.h`, `lib.h`, and `memory.h`.
- `LINE_SIZE` is defined as 160, representing the width of a line in the screen buffer.
- The `ScreenBuffer` struct is defined, which holds a pointer to the screen buffer, the current column, and the current row.
- Function declarations for `printk` and `write_screen` are provided.

In "print.c":

- The `ScreenBuffer` instance `screen_buffer` is defined, which holds the address of the screen buffer (`0xb8000` in physical memory), the current column, and the current row.
- Helper functions:
  - `udecimal_to_string`: Converts an unsigned 64-bit integer to a string representation.
  - `decimal_to_string`: Converts a signed 64-bit integer to a string representation.
  - `hex_to_string`: Converts an unsigned 64-bit integer to a hexadecimal string representation.
  - `read_string`: Copies a null-terminated string into the output buffer.
- The `write_screen` function takes a buffer, size, and color as arguments and writes the contents of the buffer to the screen buffer. It handles newline characters by advancing the row and resetting the column.
- The `printk` function is a variadic function that emulates the behavior of the `printf` function. It takes a format string and a variable number of arguments.
  - The function iterates over the characters of the format string and processes format specifiers (e.g., `%x`, `%u`, `%d`, `%s`).
  - Depending on the format specifier, the function retrieves the corresponding argument from the variable argument list using `va_arg`.
  - The function uses the helper functions to convert the argument value to a string representation and appends it to the output buffer.
  - Finally, it calls `write_screen` to display the formatted string on the screen.

- Note: The `0xf` argument passed to `write_screen` represents the color attribute of the displayed text.

Overall, the code provides a basic printing functionality for writing formatted text to the screen buffer. It supports format specifiers for hexadecimal, unsigned decimal, signed decimal, and string values.

process modules

Let's break down the code:

In "process.h":

- The header file includes the necessary dependencies: `trap.h`.
- The `Process` struct is defined, representing a process. It includes fields for the process ID (`pid`), state, page map, stack address, and a pointer to a `TrapFrame` struct.
- The `TSS` struct is defined, representing the Task State Segment. It includes various fields related to the processor's task switching mechanism.
- Constants and function declarations are provided.

In "process.c":

- The file includes the necessary dependencies: `process.h`, `trap.h`, `memory.h`, `print.h`, `lib.h`, and `debug.h`.
- The external `Tss` variable is declared, representing the Task State Segment.
- The `process_table` array is defined, which holds the process entries for a fixed number of processes (`NUM_PROC`).
- The `pid_num` variable is defined to keep track of the next process ID.
- Helper functions:
  - `set_tss`: Sets the `rsp0` field of the `Tss` struct to the specified process's stack address plus the stack size, indicating the stack pointer to be used when switching to that process's kernel stack.

    - `find_unused_process`: Searches the `process_table` array for an unused process entry (state is `PROC_UNUSED`) and returns a pointer to it.
    - `set_process_entry`: Sets up the necessary fields in a process entry, including the process state, process ID, stack allocation, `TrapFrame` initialization, page map setup, etc.
- The `init_process` function is called during system initialization. It finds an unused process entry and sets up the process using `set_process_entry`.
- The `launch` function is responsible for launching the initial process. It sets the `Tss` to use the initial process's kernel stack, switches the virtual memory to the initial process's page map using `switch_vm`, and starts the process using `pstart`.

Overall, this code sets up a simple process management system with support for initializing and launching a single process. The `Process` struct represents a process and includes fields for its state, ID, stack, page map, and `TrapFrame`. The `process_table` array holds multiple process entries, and the `Tss` struct represents the Task State Segment used for task switching. The code provides functions to find an unused process entry, set up the necessary fields for a process, initialize the process during system startup, and launch the initial process.

system calls

Let's break down the code:

In "syscall.h":

- The header file includes the necessary dependency: `trap.h`.
- A `SYSTEMCALL` function pointer type is defined, representing the type of a system call function.
- Function declarations for `init_system_call` and `system_call` are provided.

In "syscall.c":

- The file includes the necessary dependencies: `syscall.h`, `print.h`, `debug.h`, and `stddef.h`.
- The `system_calls` array is defined to hold function pointers to system call functions. It has a fixed size of 10, representing a limited number of system calls.

- The `sys_write` function is a system call implementation for writing to the screen. It takes an `argptr` parameter, which is expected to be an array of two elements: the buffer address and the buffer size. It calls the `write_screen` function from `print.h` to write the buffer contents to the screen and returns the buffer size as the result of the system call.
- The `init_system_call` function is called during system initialization. It sets up the system call table by assigning the `sys_write` function to the first entry in the `system_calls` array.
- The `system_call` function is the entry point for handling system calls. It takes a pointer to a `TrapFrame` struct as a parameter, which contains the register values at the time of the system call.
    - The system call number is stored in the `rax` register, the number of parameters in the `rdi` register, and the address of the parameter array in the `rsi` register.
    - The function checks if the number of parameters is valid and if the system call number is valid (in this case, it only checks if it's zero).
    - If the parameters are valid, it retrieves the system call function pointer from the `system_calls` array based on the system call number.
    - It then calls the corresponding system call function, passing the `argptr` as the parameter, and stores the result in the `rax` register of the `TrapFrame`.
    - If the system call number is not valid or the parameters are invalid, it sets the `rax` register to -1 to indicate an error.

Overall, this code sets up a basic system call mechanism with support for registering system call functions and handling system calls. The `SYSTEMCALL` typedef defines the function pointer type for system call functions. The `system_calls` array holds function pointers to system call implementations, and the `init_system_call` function initializes the system call table by assigning the appropriate function to each entry. The `system_call` function handles incoming system calls by extracting the system call number and parameters from the `TrapFrame` struct, validating them, calling the corresponding system call function, and returning the result in the `rax` register. In this specific code, there is only one system call implemented (`sys_write`), which writes to the screen using the `write_screen` function from `print.h`.

interrupt handling

Let's go through it step by step:

In "trap.h":

- The header file includes the necessary dependency: `stdint.h`.
- The struct `IdtEntry` represents an entry in the Interrupt Descriptor Table (IDT). It contains fields for storing information about the interrupt handler's address, selector, attributes, and other related data.
- The struct `IdtPtr` represents the IDT pointer, which consists of a limit field indicating the size of the IDT and an address field representing the starting address of the IDT.
- The struct `TrapFrame` defines the structure of the trap frame, which is used to save the register values when an interrupt or exception occurs.
- Function declarations for various interrupt and exception vectors, as well as functions for initializing the IDT, acknowledging end of interrupt (EOI), loading the IDT, and reading the Interrupt Service Routine (ISR) and Control Register 2 (CR2) are provided.

In "trap.c":

- The file includes the necessary dependencies: `trap.h`, `print.h`, and `syscall.h`.
- The `idt_pointer` variable of type `IdtPtr` is declared to hold the IDT pointer value.
- The `vectors` array of type `IdtEntry` is declared to hold the individual interrupt and exception entries of the IDT.
- The `init_idt_entry` function is a helper function that initializes an IDT entry with the provided address and attribute values.
- The `init_idt` function initializes the IDT by calling `init_idt_entry` for each vector entry, providing the appropriate address and attribute values.
- The `handler` function is the main interrupt and exception handler. It takes a pointer to a `TrapFrame` struct as a parameter, representing the saved register values during the interrupt or exception.
    - It switches on the `trapno` field of the `TrapFrame` to determine the type of interrupt or exception.

- If the interrupt is IRQ 0 (32), it calls the `eoi` function to acknowledge the end of the interrupt.
- If the interrupt is IRQ 7 (39), it reads the ISR and checks if bit 7 is set, indicating a spurious interrupt. If so, it calls `eoi` to acknowledge the end of the interrupt.
- If the interrupt is interrupt 0x80, it calls the `system_call` function from `syscall.h` to handle system calls.
- If none of the above cases match, it prints an error message containing information about the trap number, the privilege level (ring) at which the trap occurred, the error code (if any), the value of CR2 (used for page faults), and the RIP (instruction pointer) where the trap occurred. After printing the error message, it enters an infinite loop, effectively halting the system.

Overall, this code sets up the Interrupt Descriptor Table (IDT) with entries for various interrupts and exceptions. It also defines a trap handler function that is called when interrupts or exceptions occur. The handler differentiates between different types of interrupts and exceptions and performs appropriate actions, such as acknowledging the end of interrupts, handling spurious interrupts, and invoking system calls. It also includes error handling and printing for unhandled interrupts or exceptions.

lib.h

Here's an explanation of the code:

- The header file guards (`#ifndef`, `#define`, and `#endif`) are used to prevent multiple inclusions of the same header file.
- The function prototype `int printf(const char *format, ...)` declares a variadic function named `printf`. This function is commonly used for formatted output and is typically found in standard libraries.
  - The `const char *format` parameter is a string that specifies the format of the output.
  - The `...` indicates that the function can accept a variable number of arguments after the `format` parameter. This allows the function to handle different types and quantities of arguments based on the format specifier provided.

- By declaring this function prototype in the "lib.h" header file, other source files that include this header can use the `printf` function without needing to define it again. Instead, they can rely on an implementation of `printf` provided by a library or the standard C library.

In summary, the "lib.h" header file provides a function prototype for `printf`, allowing other source files to use this function for formatted output.

syscall.asm

Here's an explanation of the code:

- The code defines a function named `writeu` in the `.text` section. The `global` directive is used to make the function visible to other parts of the program.
- Inside the function, the stack is adjusted by subtracting 16 bytes (`sub rsp,16`). This is done to allocate space for local variables or to save register values.
- The `xor eax,eax` instruction clears the `eax` register by XORing it with itself, effectively setting it to zero. The `eax` register is commonly used as the return value register in x86 calling conventions.
- The next two instructions (`mov [rsp],rdi` and `mov [rsp+8],rsi`) store the values of the `rdi` and `rsi` registers on the stack. These instructions save the function arguments passed to `writeu` so that they can be used later.
- The following instructions (`mov rdi,2` and `mov rsi,rsp`) set the `rdi` and `rsi` registers with the appropriate values for making a system call.
    - `rdi` is set to 2, which typically represents the system call number for writing to a file descriptor (stdout).
    - `rsi` is set to the address in memory where the function arguments are stored (previously saved on the stack).
- The `int 0x80` instruction triggers a software interrupt, specifically interrupt 0x80. In x86 assembly, this is a common way to invoke system calls. The values in `rdi` and `rsi` are used to specify the system call and its arguments.
- After the system call, the stack is restored (`add rsp,16`) to its original position, and the function returns (`ret`).

Overall, the `writeu` function appears to be a wrapper or helper function for making a system call to write data to a file descriptor, possibly representing standard output (stdout).

start.asm

Here's an explanation of the code:

- The code defines a label named `start` in the `.text` section. The `global` directive is used to make the label visible to other parts of the program.
- The `extern` directive is used to declare the symbol `main` as an external symbol. This indicates that `main` is defined in a different module or file and will be linked later.
- Inside the `start` block, the `call main` instruction is used to call the `main` function. This is the entry point of the program, where the execution starts.
- After calling `main`, the `jmp $` instruction is used to perform an infinite loop. `jmp $` jumps to the current instruction, effectively creating an endless loop. This ensures that the program does not exit and continues executing indefinitely.

The purpose of this code is to provide a simple program entry point and create an infinite loop after calling the `main` function.

redefined printf function:

Here's an explanation of the code:

- The function `writeu` is declared as an external function. It is assumed to be implemented elsewhere, and it is responsible for writing the contents of a buffer to a specified output.
- The function `udecimal_to_string` converts an unsigned decimal number to a string representation. It iteratively extracts each digit of the number by taking the modulus of 10, converts it to the corresponding character, and stores it in a buffer. The digits are then copied to the main buffer in reverse order.

- The function `decimal_to_string` converts a signed decimal number to a string representation. It first checks if the number is negative and adds a negative sign to the buffer if so. Then it calls `udecimal_to_string` to convert the absolute value of the number to a string.
- The function `hex_to_string` converts an unsigned integer to a hexadecimal string representation. It follows a similar process as `udecimal_to_string`, but using base 16 instead of base 10. It also appends the 'H' character to indicate the string represents a hexadecimal value.
- The function `read_string` copies a null-terminated string to the main buffer, starting from the specified position.
- The `printf` function is defined to accept a format string and variable arguments. It initializes a buffer to store the formatted output and keeps track of the buffer size. It uses a `va_list` to access the variable arguments.
- The `printf` function iterates over each character in the format string. If a character is not a '%' character, it is copied directly to the buffer. If a '%' character is encountered, the next character is checked to determine the format specifier. Depending on the specifier, the corresponding argument is extracted from the variable arguments using `va_arg`. The appropriate conversion function is called, and the resulting string is appended to the buffer.
- After formatting the output, the `printf` function calls `writeu` to write the buffer to the output. The `buffer_size` is returned as the result.

Overall, this code provides a basic implementation of the `printf` function that supports conversion specifiers for unsigned and signed integers, hexadecimal values, and strings. The formatted output is stored in a buffer and then passed to an external `writeu` function for actual output.

main.c and scheduler.c files

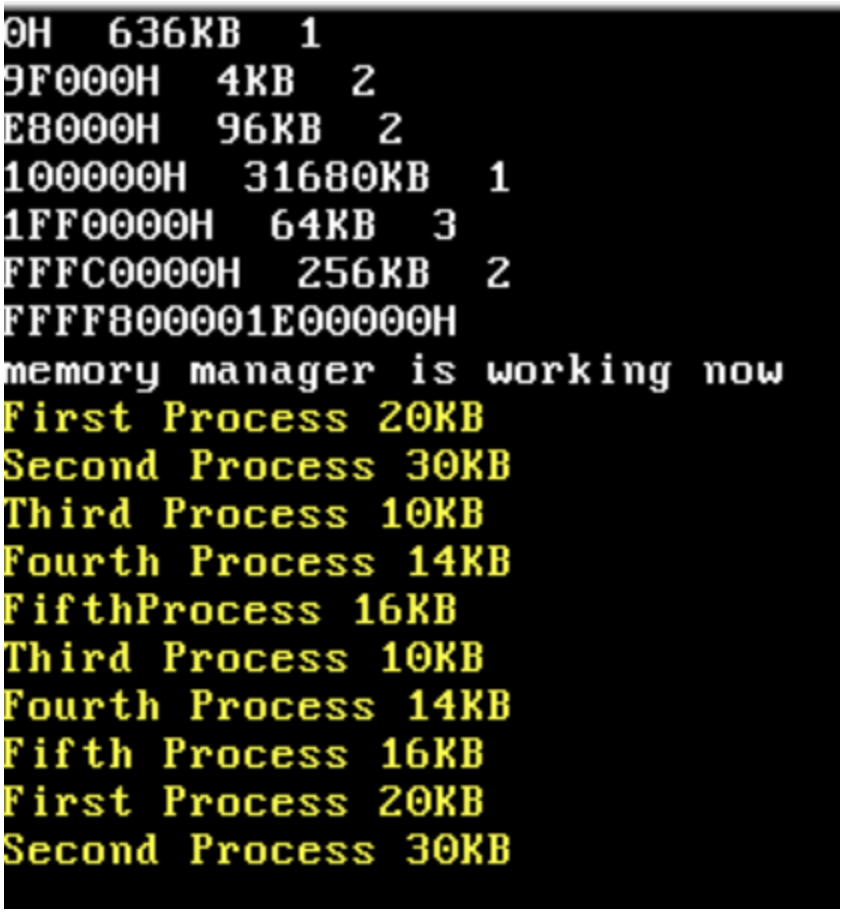
A scheduler is a crucial component of an operating system that is responsible for managing the execution of processes or threads on a computer system's CPU (Central Processing Unit). Its main objective is to allocate system resources efficiently and fairly among multiple processes or threads, ensuring optimal system performance and responsiveness.

The primary tasks of a scheduler include:

1. Process/Thread Scheduling: The scheduler determines which processes or threads should be executed and in what order. It assigns CPU time slices (time intervals) to each process or thread, allowing them to execute for a certain period before switching to another process or thread. This switching is known as context switching.
2. Priority Management: The scheduler may assign priorities to processes or threads based on factors such as their importance, resource requirements, or user-defined criteria. Higher-priority processes or threads are given preferential treatment in terms of CPU allocation, allowing critical tasks to be executed promptly.
3. Fairness and Resource Allocation: The scheduler ensures fair distribution of CPU time among processes or threads to prevent monopolization by a single process. It aims to provide an equitable share of CPU resources to all active processes or threads based on their priorities or other defined criteria.
4. Synchronization and Coordination: The scheduler synchronizes the execution of multiple processes or threads, ensuring that they do not interfere with each other or access shared resources simultaneously. It coordinates resource access, manages locks, and enforces synchronization mechanisms to maintain data integrity and avoid conflicts.
5. Deadlock Prevention: The scheduler may employ techniques to prevent or resolve deadlocks, which occur when processes or threads are unable to proceed because they are waiting for resources held by other processes or threads. Deadlock prevention algorithms analyze resource dependencies and take actions to avoid or break deadlocks.
6. Power Management: In systems with power-saving features, the scheduler may optimize power consumption by selectively suspending or reducing the frequency of execution for certain processes or threads when their workload or priority allows it. This helps conserve energy and extend battery life in mobile devices.

Overall, the scheduler plays a crucial role in maximizing system throughput, ensuring fairness, responsiveness, and efficient resource utilization. Different scheduling algorithms exist, such as round-robin, priority-based, and multi-level feedback queue, each with its own characteristics and suitability for specific system requirements.

Here we have done a scheduler which utilizes non-preemptive SJF algorithm.



That’s why the processes shown above are executed in sequence and not preempted.

CONCLUSION:

Detailed summary of the project so far

1. syscall.h and syscall.c:
    - The `syscall.h` header file defines the system call interface and function prototypes.
    - The `syscall.c` file implements the system call functionality.

- The `init_system_call` function initializes an array of system call function pointers.
- The `system_call` function handles the system call interrupt, retrieves the system call number and arguments from the TrapFrame structure, and invokes the corresponding system call function.

2. trap.h and trap.c:
   - The `trap.h` header file defines structures related to the Interrupt Descriptor Table (IDT) and TrapFrame.
   - The `trap.c` file implements functions related to IDT initialization and trap handlers.
   - The `init_idt` function initializes entries in the IDT for various interrupts and traps.
   - The `handler` function is the main trap handler that handles different types of traps, including interrupt 32 (timer interrupt), interrupt 39 (IRQ7 interrupt), interrupt 0x80 (system call interrupt), and default traps.
3. lib.h and lib.c:
   - The `lib.h` header file declares the `printf` function prototype.
   - The `lib.c` file implements the `printf` function, which supports formatting and printing of various data types.
   - The `printf` function processes the format string and handles placeholders for integers (`%d` and `%u`), hexadecimal values (`%x`), and strings (`%s`) using helper functions such as `decimal_to_string`, `udecimal_to_string`, `hex_to_string`, and `read_string`.
   - The formatted output is stored in a buffer and then passed to the `writeu` function to write it to the screen.
4. main.c:
   - The `main` function is the entry point of the program.
   - It calls the `printf` function to print messages indicating the sizes of different processes.

- It then calls the `schedule` function, which prints the sizes of processes in a different order.

In summary, the provided codes include the implementation of system calls, trap handling, a simple printing library, and a demonstration of process scheduling. The system call functionality allows interaction between user programs and the underlying operating system. The trap handling and IDT initialization ensure proper handling of various interrupts and traps. The printing library provides a basic `printf` function for formatted output. Finally, the `main` function demonstrates a sample program that prints process sizes and showcases a simplistic scheduling sequence.

FUTURE PLANS:

Here are some potential future plans or enhancements that could be considered:

1. Implement File System: One possible future plan could be to add file system support to the operating system. This would involve designing and implementing file data structures, file operations (such as create, read, write, delete), directory management, and disk I/O.
2. Expand System Calls: The project could be expanded by adding more system calls to provide additional functionality to user processes. Examples could include file-related system calls (open, close, seek) or process management system calls (fork, exec).
3. Memory Management: Enhancing the project to include memory management would be a valuable addition. This could involve implementing mechanisms such as virtual memory, paging, memory allocation algorithms (e.g., malloc), and handling memory protection.
4. Multitasking and Scheduling Policies: Currently, the project includes a simple process scheduling mechanism. Future plans could involve implementing more advanced scheduling policies, such as priority-based scheduling, round-robin scheduling, or preemptive scheduling, to improve system performance and fairness.
5. User Interface and Shell: Developing a user interface and a shell environment would provide a more interactive experience for users. This could include

command-line parsing, executing user commands, and supporting features like input/output redirection and pipes.

6. Device Drivers: Introducing device drivers would allow the operating system to interact with hardware devices, such as keyboards, displays, or network interfaces. Implementing drivers would involve handling hardware interrupts, managing device resources, and providing appropriate abstractions for user processes to access devices.
7. Error Handling and Robustness: Enhancing error handling mechanisms and adding robustness to the system would be essential for a production-grade operating system. This could involve error code propagation, exception handling, and fault tolerance mechanisms.

These are just a few potential future plans for the project. The actual direction and priorities would depend on the specific goals and requirements of the project, as well as the expertise and resources available for its development.