Phase 1 - Kernel Bootstrap

Due Oct 9 by 11:59pm **Points** 100

Phase 1 - The Kernel Bootstrap

Project Repositories

Each team member must accept the assignment on GitHub Classroom:

- **https://classroom.github.com/g/0y7Oi0Cj (https://classroom.github.com/g/0y7Oi0Cj)**

When accepting the assignment, select your team from the list, or create your team if it is not yet listed.

Each team will have a repository that team members will clone from, commit changes, and collaborate together.

Project Phase Submissions

Submissions must be on the main branch (teams are welcome to create other branches for individual team purposes) but ONLY the main branch will be considered for submissions.

Submissions will be based upon a git tag for each project phase, example:

- phase-1.0

Future project phases would be tagged as such:

- phase-2.0

You may re-tag any commit as a submission. If you have additional submissions PAST the due date with corrections to be made, you may tag them with a sub version, such as:

- phase-1.1
- phase-1.2
- …
- phase-2.1
- etc.

Phase 1 Goals and Requirements

In this project phase, we will develop our kernel bootstrap that will perform the following:

- Initialize Kernel Data Structures
- Initialize the Interrupt Descriptor Table (IDT)
- Schedule Processes
- Load Process(es)
- Process Interrupts

From the operating system bootstrap template, all functions which contain the following comment:

/* !!! CODE NEEDED !!! */ must be completed per the specifications described, and following the concepts outlined in this guide.

File Layout/Organization

The initial repository contains a template for the operating system bootstrap. It is buildable and runnable, but still requires implementation for functional completeness.

Filename	Scope/Context	Description
main.c	Kernel	Main entry point Initializes kernel data structures Initializes IDT Launches idle task Starts the process scheduler Loads the initial task Contains the kernel run loop
spede.h	Kernel/User	Includes SPEDE headers
global.h	Kernel/User	Global definitions available for both kernel and user space
kernel.h	Kernel	Kernel data structures Kernel helper functions (panic, panic_warn, debug_printf)
kernel.c	Kernel	Kernel helper functions implementations
kisr.h	Kernel	Interrupt Service Routine definitions
kisr.c	Kernel	Interrupt Service Routine implementation
kisr_entry.S	Kernel	Interrupt Service Routine assembly functions
kproc.h	Kernel	Process Management definitions Process scheduler Process execution Process exit *Kernel idle task

kproc.c	Kernel	Process Management implementation
kproc_entry.S	Kernel	Process Management assembly functions
user_proc.h	User	User Process(es) definitions
user_proc.c	User	User Process(es) implementation
string.h	Kernel/User	String utilities declarations
string.c	Kernel/User	String utilities implementation
queue.h	Kernel/User	Queue data type definition Queue function declarations
queue.c	Kernel/User	Queue function implementations

Project Phase 1 Implementation

Data types and structures

Enumerations

- Process States

Data Types

- Queues
- Process Control Block

Data Structures

- Process queues
- Stack
- Process Control Blocks
- System Time
- Running Process ID

Utility Functions

Part of implementing our operating system requires the implementation of various "utility" functions that facilitate manipulation of data types and data structures.

Your team must implement the functions to support the following:

- Enqueuing and dequeueing integer values from a simple integer-based queue
- Performing string/byte array handling

Kernel Data Structure Initialization

When the kernel loads, memory, variables, and data structures need to be initialized to known/default values. You cannot assume that a variable or data structure contains a value unless you have specifically initialized it.

Interrupt Descriptor Table

In this first project phase, the only interrupt that will be enabled is the timer interrupt. This interrupt (timer channel 0) is driven by the Programmable Interval Timer (PIT) that is driven by a 1193182Hz signal that can be programmed to "tick" at different frequencies (between 18.2065Hz and 1193182Hz). In the SPEDE environment, the timer is configured to operate at 100Hz, which means it will trigger 100 times per second.

This timer is the basis of performing process scheduling.

The interrupt timer must be enabled by:

- Adding an entry to the IDT (contains the interrupt number and pointer to the handler)
- Clearing the interrupt mask

In future project phases, additional interrupts will be enabled in the same manner.

Interrupt Processing

In our operating system kernel, interrupt processing is composed of the following steps:

- Detecting the interrupt that was triggered
- Passing control to the kernel run loop
- Kernel run loop dispatches to the interrupt service routine
- Interrupt service routine runs
- Interrupts are re-enabled (for hardware interrupts)
- Process loading occurs to restore the previous process state

When an interrupt occurs, the following operations take place (these happen "transparently" by the CPU)

- Save the state of the interrupted procedure (register values pushed to stack)
- Save the previous data segment (pushed to stack)
- Load the new instruction pointer for the interrupt service routine
- Pass control to the interrupt handler

When the handler has completed, the opposite must be performed:

- Restore the data segment
- Restore the state of the interrupted procedure

In our kernel, the restoration is performed in the process loading operations as part of the kernel run loop.

Scheduling a Process

In our bootstrapped operating system kernel, a process may be defined in three distinct states:

- Available
- Ready
- Running

Processes will exist in three different locations representing each state:

- Available queue: a queue of available process ids
- Run queue: a queue of processes that are ready to be run
- Running: a process is actively running

The process scheduling implementation coordinates the activities of moving processes between the various queues and the actively running state.

At boot, processes do not yet exist, and are added to the "Available" queue during kernel initialization. Processes must be executed to be added to the run queue, and processes must exit to move back into the available queue.

When a process is executed (see details below) it will be dequeued from the available queue and queued into the run queue.

As the timer interrupt is triggered, each tick increments the following:

- The system time
- The process run time

When the process run time has exceeded the process time slice (maximum number of ticks that a process can run) it will be unscheduled by queuing to the run queue, and resetting the running PID.

When the process scheduler is triggered, it will:

- Dequeue a process from the run queue
- Set the running PID for the process

The kernel run loop will then load the process.

Executing a process

When a process is executed, all of the process data structures must be initialized:

- Process control block
- Trapframe (initial CPU state)
- Enqueue the process to the run queue

Loading a Process

A process that is scheduled must be loaded. This takes place by restoring the CPU state via the trapframe stored on the stack.

Unloading (exiting) a Process

Since the process execution routine initializes all data structures to known values, the unload routine can simply move the active process to the available queue and then trigger the process scheduler.