CpS 230: Computer Systems

Team Project: Kernel

by

Zachary Hayes

Ryan Longacre

December 6, 2017

# Overview

This project demonstrates how an operating system’s kernel works by writing a multitasking program for DOS that runs 4 or more tasks simultaneously. It is written completely in 16-bit DOS Assembly. We use many DOS video interrupts, hook the BIOS keyboard interrupt, hook the timer interrupt, use an elapsed time wait interrupt, use direct port output, and display graphics by writing directly to the PC text graphics memory map. Our kernel’s tasks demonstrate graphics, animation, expression calculation, and sound output. We also write a bootloader to load the kernel into memory and run it using the BOOT command in DOSBox.

# Results

## Requirements Implementation Summary

The following requirements and/or electives have been implemented:

* Multi-tasking support for at least 4 tasks
* Each task has 256 bytes of dedicated stack space
* Bootloader with Master Boot Record signature that loads the kernel into memory and runs it
* Two scrolling text graphics tasks
* Bouncing ball animation graphics task
* Conway’s Game Of Life graphics task
* Reverse Polish Notation Calculator task
* Music task

## Bug Summary

1. If you look closely, the ball animation briefly erases one of Task A or Task B’s characters when it moves over the string. However, the strings are redrawn the next time the task runs.
2. Not all valid keypresses are registered by the RPN Calculator. One cannot type quickly and have all the characters acknowledged.

# Details

## Multitasking Logic

Each task is given a 256-byte “stack” and a stack pointer and is loaded by calling \_spawn\_new\_task before the \_main function runs. \_spawn\_new\_task looks for an unused task stack and when it finds one pushes the task’s registers and flags on to its “stack,” as well as a fake return address. It then marks the task as active so future spawns don’t overwrite the task. After spawning all of the tasks, the \_main function runs first and then yields, starting the multitasking. When a task is finished running, it calls \_yield, which pushes the task’s flags and registers, then switches the active stack and stack pointer to the next task. \_yield then returns to the next task, which runs, \_yields, and the cycle of apparent “multitasking” continues.

## Bootloader Logic

The bootloader performs the necessary segment and memory setup tasks such as setting DS to CS and SS and SP to 0x0000. It then prints a message with the class number, project name, and our names. It then uses BIOS keyboard interrupt 0x16,0 to wait for a keypress before booting. Once a key is pressed, the bootloader loads 13 sectors, starting with sector 2, from the bootdisk.img floppy disk image file. When the sectors have finished loading, our kernel runs.

## Task A/Task B strings

Prints a text string in the Graphics window of the kernel and scrolls it left and right in the window as the task runs consecutively. Task A prints “I am task A” and Task B prints “I am**\r\n**task B.” This task uses helper functions we wrote \_printString and \_printChar, and Task B’s string in particular tests the line ending capabilities of \_printString. These functions take various location, color, and string/character parameters and modifies the text graphics memory map accordingly. In order to make the strings change direction when they reach a border of the window, we compare the BL register (which holds the string’s column data times 2) to hardcoded values representing the borders of the Graphics window. We use a global variable for each task to store the direction each string is moving. To prevent the strings from “smearing”—not erasing the first or last character when moving right or left—the tasks first reprint the string in black foreground before moving them to “erase” them.

## Ball animation

A simple generalization of the Task A/Task B strings. Prints a space character (‘ ‘) as the “ball,” which moves through the Graphics window and bounces off the borders. Uses the same methods of border checking, direction changing, and erasing as the Task A/B strings do, with a little extra code to work vertically as well as horizontally. Because the “ball” is only one character, some minor differences are visible in the code to only use the \_printChar function.

## Conway’s Game of Life

Displays a “glider” pattern in the Game of Life window which moves according to the generation rules in John Conway’s Game of Life. Live cells are displayed as red asterisk characters (‘\*’), while dead cells are spaces (‘ ’). This version of the Game of Life is computed on a torus-shaped board, which allows cells to wrap around from left to right and top to bottom. The board is 20 cells wide by 10 cells tall. The boards holding the current generation and the one required to compute the next generation are stored in memory in an array of bytes.

## RPN Calculator

Computes Reverse Polish Notation (also called postfix) expressions typed with the keyboard. Only characters that are legal in RPN expressions are acknowledged: numbers; number pad numbers; operators +, -, \*, /, ~, and %; and spaces. Supports addition, subtraction, multiplication, division, negation (using the ~ operator with one operand), and modulus (using the % operator). Answers are stored in 16-bit integers, so overflow can occur for answers (intermediate or final) larger than 65,535 or less than -65,536. The RPN stack can hold up to 16 values; should more values be required, the calculator returns a “Stack overflow!” error message. In case of malformed expressions, the calculator returns a “Stack underflow!” or “Divide by 0!” (appropriately) error message. Supports multi-digit numbers in expressions, so numbers must be terminated by non-number characters (For example, 33+ is interpreted as 33 + ?, which would result in stack underflow. However, 3 3+ is interpreted as 3 + 3 and would return 6.). Uses global variables for current input string, last expression entered and its result, current number being processed, stack of operands, and pointer to top of operand stack. Expressions are evaluated when Enter is pressed. When not evaluating an expression, the calculator task is responsible for printing the current expression being entered and moving DOS’s cursor.

## Music

Plays a tune through the computer speakers. Direct port input and output was used to interface with the computer speaker. The timer interrupt was hooked to allow for precise timing of note lengths as well as still allowing for multithreading. The music note frequencies are stored in an array of two-byte numbers, and their respective frequencies are stored in an array of bytes.

# Contributions

## Zachary Hayes

Time spent: 20.0 hrs.

Contributions:

* Animation pacing (elapsed time interrupt)
* Border checking with Tasks A and B
* Bootloader
* Visual layout of kernel (wrote code to draw it)
* Keyboard interrupt hook, key processing
* RPN Calculator
* Bouncing ball animation
* Wrote most of project report

## Ryan Longacre

Time spent: 19.5 hrs.

Contributions:

* \_printChar and \_printString
* Fixed screen flicker
* Conway’s Game of Life task
* Music task

# Appendix *– Complete code listing*

## kernel.asm

; CpS 230 Team Project - Kernel

; Zachary Hayes (zhaye769) and Ryan Longacre (rlong315)

bits 16

org 0x0 ; change to 0x100 when running as COM file, change to 0x0 when booting with bootloader

SECTION .text

start:

mov dx, cs

mov ds, dx

; set up custom hardware interrupts

; starting with the keyboard

cli

mov ax, 0

mov es, ax

mov dx, [es:0x9\*4]

mov [previous9], dx

mov ax, [es:0x9\*4+2]

mov [previous9+2], ax

mov dx, keyboard

mov [es:0x9\*4], dx

mov ax, cs

mov [es:0x9\*4+2], ax

; then the timer

mov ax, 0

mov es, ax

mov dx, [es:0x8\*4]

mov [previous8], dx

mov ax, [es:0x8\*4+2]

mov [previous8+2], ax

mov dx, timer

mov [es:0x8\*4], dx

mov ax, cs

mov [es:0x8\*4+2], ax

sti

; spawn tasks

mov dx, \_main

call \_spawn\_new\_task

mov dx, \_taskA

call \_spawn\_new\_task

mov dx, \_taskB

call \_spawn\_new\_task

mov dx, \_ballTask

call \_spawn\_new\_task

mov dx, \_rpnCalculator

call \_spawn\_new\_task

mov dx, \_gameOfLife

call \_spawn\_new\_task

mov dx, \_task\_Music

call \_spawn\_new\_task

jmp \_main

terminate:

; restore old hardware interrupts

; starting with the keyboard

mov ax, 0

mov es, ax

mov dx, [previous9]

mov [es:0x9\*4], dx

mov ax, [previous9 + 2]

mov [es:0x9\*4+2], ax

; then the timer

mov ax, 0

mov es, ax

mov dx, [previous8]

mov [es:0x8\*4], dx

mov ax, [previous8 + 2]

mov [es:0x8\*4+2], ax

; stop the current note that's playing

call \_stopNote

mov ah, 0x4c

mov al, 0

int 0x21

; terminates program; exit code 0

; spawns new task

; dx should contain the address of the function to run

\_spawn\_new\_task:

; save current stack pointer

mov bx, stack\_pointers

add bx, [current\_task]

add bx, [current\_task] ; add twice because we have two bytes

mov [bx], sp

; switch to new stack

mov cx, 0

mov cl, [current\_task]

inc cl

sp\_loop\_for\_available\_stack:

cmp cl, byte [current\_task]

jne sp\_check\_for\_overflow

jmp sp\_no\_available\_stack

sp\_check\_for\_overflow:

cmp cl, 7

jg sp\_reset

jmp sp\_check\_if\_available

sp\_reset:

mov cl, 0

jmp sp\_loop\_for\_available\_stack

sp\_check\_if\_available:

mov bx, task\_status

add bx, cx

cmp byte [bx], 0

je sp\_is\_available

inc cx

jmp sp\_loop\_for\_available\_stack

sp\_is\_available:

mov bx, task\_status

add bx, cx

mov byte [bx], 1

; push a fake return address

mov bx, stack\_pointers

add bx, cx

add bx, cx

mov sp, [bx]

push dx

; push registers

pusha

; push flags

pushf

; update stack pointer for task

mov bx, stack\_pointers

add bx, cx

add bx, cx ; add twice because we have two bytes

mov [bx], sp

; restore to original stack

sp\_no\_available\_stack:

mov bx, stack\_pointers

add bx, [current\_task]

add bx, [current\_task] ; add twice because we have two bytes

mov sp, [bx]

ret

; yields processor from caller to next task

\_yield:

pusha ; push registers

pushf ; push flags

; save current stack pointer

mov bx, stack\_pointers

add bx, [current\_task]

add bx, [current\_task] ; add twice because we have two bytes

mov [bx], sp

; switch to new stack

mov cx, 0

mov cl, [current\_task]

inc cl

y\_check\_for\_overflow:

cmp cl, 7

jg y\_reset

jmp y\_check\_if\_enabled

y\_reset:

mov cl, 0

jmp y\_check\_for\_overflow

y\_check\_if\_enabled:

mov bx, task\_status

add bx, cx

cmp byte [bx], 1

je y\_task\_available

inc cx

jmp y\_check\_for\_overflow

y\_task\_available:

mov bx, cx

mov [current\_task], bx

; update stack pointer

mov bx, stack\_pointers

add bx, [current\_task]

add bx, [current\_task] ; add twice because we have two bytes

mov sp, [bx]

; pop flags

popf

; pop registers

popa

ret

; Plays a song with interrupts

\_task\_Music:

push ax

push bx

push cx

mov ax, [isNotePlaying]

cmp ax, 0

je \_not\_playing

jmp \_yield\_music

\_not\_playing:

mov bx, [note\_pointer]

mov cx, 0

mov cl, byte [song\_notes\_duration + bx]

add bx, bx

mov al, byte [song\_notes\_pitch + bx]

inc bx

mov ah, byte [song\_notes\_pitch + bx]

mov bx, ax

mov ax, word [note\_pointer]

inc ax

cmp ax, 42

jl \_skip

mov ax, 0

\_skip:

mov word [note\_pointer], ax

call \_playNote

\_yield\_music:

pop cx

pop bx

pop ax

call \_yield

jmp \_task\_Music

; Plays a note through the computer speaker

; note is stored in bx

; duration is stored in cx

\_playNote:

push ax

mov word [isNotePlaying], 1

cmp bx, 0

jne \_notWait

mov [noteDuration], cx ; Duration of the rest

pop ax

ret

\_notWait:

mov al, 182 ; Prepare the speaker for the

out 43h, al ; note.

mov ax, bx ; Frequency number

mov [noteDuration], cx ; Duration of the note

out 42h, al ; Output low byte.

mov al, ah ; Output high byte.

out 42h, al

in al, 61h ; Turn on note (get value from port 61h).

or al, 00000011b ; Set bits 1 and 0.

out 61h, al

pop ax

ret

; Stops the computer speaker

\_stopNote:

push ax

mov word [isNotePlaying], 0

in al, 61h ; Turn on note (get value from port 61h).

and al, 11111100b ; Set bits 1 and 0.

out 61h, al

pop ax

ret

; Prints "Task A" to screen

\_taskA:

mov bl, 44

jumpA\_begin:

; print the string with black foreground to effectively erase

mov bh, 34

mov cl, 0 ; black background

mov ch, 0 ; black foreground

mov ax, taskA\_str ; pointer to string

mov dh, 0 ; no blink

call \_printString

; increment column for this print

cmp bl, 160 - 22 ; 22 is length of string

jae changeDir\_A

cmp bl, 42

jbe changeDir\_A

cmp byte [taskA\_dir], 1 ; check direction flag

je moveRight\_A

; move left

sub bl, 2

jmp print\_A

moveRight\_A:

add bl, 2

jmp print\_A

changeDir\_A:

mov al, 4 ; these next 4 lines evaluate to bh -= ((dir == 1) ? 2 : -2)

imul byte [taskA\_dir]

add al, -2

sub bl, al

xor byte [taskA\_dir], 1 ; flip direction flag

print\_A:

; print "Task A" in light blue

mov bh, 34

mov cl, 0 ; black background

mov ch, 9 ; light blue foreground

mov ax, taskA\_str ; pointer to string

mov dh, 0 ; no blink

call \_printString

call \_yield

jmp jumpA\_begin

; Prints "I am task B" to screen

\_taskB:

mov bl, 160 - 12 - 2 ; 12 is length of string's longest line

jumpB\_begin:

; print the string with black foreground to effectively erase

mov bh, 40

mov cl, 0 ; black background

mov ch, 0 ; black foreground

mov ax, taskB\_str ; pointer to string

mov dh, 0 ; no blink

call \_printString

; increment column this print

cmp bl, 160 - 12 ; again, 12 is length of string's longest line

jae changeDir\_B

cmp bl, 42

jbe changeDir\_B

cmp byte [taskB\_dir], 1 ; check direction flag

je moveRight\_B

; move left

sub bl, 2

jmp print\_B

moveRight\_B:

add bl, 2

jmp print\_B

changeDir\_B:

mov al, 4 ; these next 4 lines evaluate to bh -= ((dir == 1) ? -2 : 2)

imul byte [taskB\_dir]

add al, -2

sub bl, al

xor byte [taskB\_dir], 1 ; flip direction flag

print\_B:

; print "Task B" in light purple

mov bh, 40

mov cl, 0 ; black background

mov ch, 0xd ; light purple foreground

mov ax, taskB\_str ; pointer to string

mov dh, 0 ; no blink

call \_printString

call \_yield

jmp jumpB\_begin

; Prints 1-char size "ball" that bounces off window sides

\_ballTask:

mov bl, 100

mov bh, 32

jumpBall\_begin:

; print ball with black background to effectively erase

mov cl, 0 ; black background

mov ch, 0 ; black foreground

mov dl, " " ; ball char

mov dh, 0 ; no blink

call \_printChar

sub bl, 2 ; printChar increments BL, sub to avoid confusion

; increment column for this print

cmp bl, 160 - 2 ; 22 is length of string

jae changeDirX\_ball

cmp bl, 42

jbe changeDirX\_ball

cmp byte [ball\_dirX], 1 ; check direction flag

je moveRight\_ball

; move left

sub bl, 2

jmp ball\_changeRow

moveRight\_ball:

add bl, 2

jmp ball\_changeRow

changeDirX\_ball:

mov al, 4 ; these next 4 lines evaluate to bh -= ((dir == 1) ? 2 : -2)

imul byte [ball\_dirX]

add al, -2

sub bl, al

xor byte [ball\_dirX], 1 ; flip direction flag

ball\_changeRow:

; increment row for this print

cmp bh, 50 - 2 ; 22 is length of string

jae changeDirY\_ball

cmp bh, 30

jbe changeDirY\_ball

cmp byte [ball\_dirY], 1 ; check direction flag

je moveUp\_ball

; move down

sub bh, 2

jmp printBall

moveUp\_ball:

add bh, 2

jmp printBall

changeDirY\_ball:

mov al, 4 ; these next 4 lines evaluate to bh -= ((dir == 1) ? 2 : -2)

imul byte [ball\_dirY]

add al, -2

sub bh, al

xor byte [ball\_dirY], 1 ; flip direction flag

printBall:

; print ball in green

mov cl, 2 ; green background

mov ch, 0 ; black foreground

mov dl, " " ; ball char

mov dh, 0 ; no blink

call \_printChar

sub bl, 2 ; printChar increments BL, sub to avoid confusion

call \_yield

jmp jumpBall\_begin

; task that prints a 20x10 version of Conway's Game of Life

\_gameOfLife:

push ax

push bx

push cx

push dx

; set the position to start printing at 0, 30

mov bl, 0

mov bh, 30

; print and update the grid

mov ax, 0

\_y\_loop:

mov dh, 0

\_x\_loop:

; put the character to be printed into dl

push bx

mov bx, ax

mov dl, [ds:gameOfLife\_grid1+bx]

pop bx

; set the colors

mov cl, 0

mov ch, 4

; turn off blink

push dx

mov dh, 0

; print the character

call \_printChar

pop dx

; check the surrounding cells and update the current cell accordingly

push bx

call \_check\_adjacent\_cells

; if the cell has:

; less than 2 neighbors, it dies

; 2 neighbors, it stays the same

; 3 neighbors, it is set to live

; more than 3, it dies

cmp bx, 2

jl \_clear\_cell

je \_carry\_cell

cmp bx, 3

je \_set\_cell

jg \_clear\_cell

\_clear\_cell:

mov bx, ax

mov byte [ds:gameOfLife\_grid2+bx], ' '

jmp \_done\_with\_cell

\_set\_cell:

mov bx, ax

mov byte [ds:gameOfLife\_grid2+bx], '\*'

jmp \_done\_with\_cell

\_carry\_cell:

mov bx, ax

push dx

mov dl, [ds:gameOfLife\_grid1+bx]

mov byte [ds:gameOfLife\_grid2+bx], dl

pop dx

\_done\_with\_cell:

pop bx

; increment the counting registers

inc dh

inc ax

cmp dh, 20

je \_x\_loop\_end

jmp \_x\_loop

\_x\_loop\_end:

; print a CRLF character to return to the next line

inc bh

inc bh

mov bl, 0

cmp ax, 200

je \_y\_loop\_end

jmp \_y\_loop

\_y\_loop\_end:

mov bx, 0

move\_loop:

; move the character from the second grid to the first

mov dl, [ds:gameOfLife\_grid2+bx]

mov byte [ds:gameOfLife\_grid1+bx], dl

inc bx

cmp bx, 200

je move\_loop\_end

jmp move\_loop

move\_loop\_end:

; de-clobber the registers

pop dx

pop cx

pop bx

pop ax

call \_yield

jmp \_gameOfLife

; helper method for Game of Life

; takes the cell location in ax

; and returns the number of adjacent live cells in bx

\_check\_adjacent\_cells:

push ax

push dx

mov bx, ax

mov ax, 0

\_topleft:

sub bx, 21

call \_correct\_cell\_location

mov dl, byte [ds:gameOfLife\_grid1+bx]

cmp dl, ' '

je \_topcenter

inc ax

\_topcenter:

add bx, 1

call \_correct\_cell\_location

mov dl, byte [ds:gameOfLife\_grid1+bx]

cmp dl, ' '

je \_topright

inc ax

\_topright:

add bx, 1

call \_correct\_cell\_location

mov dl, byte [ds:gameOfLife\_grid1+bx]

cmp dl, ' '

je \_left

inc ax

\_left:

add bx, 18

call \_correct\_cell\_location

mov dl, byte [ds:gameOfLife\_grid1+bx]

cmp dl, ' '

je \_right

inc ax

\_right:

add bx, 2

call \_correct\_cell\_location

mov dl, byte [ds:gameOfLife\_grid1+bx]

cmp dl, ' '

je \_bottomleft

inc ax

\_bottomleft:

add bx, 18

call \_correct\_cell\_location

mov dl, byte [ds:gameOfLife\_grid1+bx]

cmp dl, ' '

je \_bottomcenter

inc ax

\_bottomcenter:

add bx, 1

call \_correct\_cell\_location

mov dl, byte [ds:gameOfLife\_grid1+bx]

cmp dl, ' '

je \_bottomright

inc ax

\_bottomright:

add bx, 1

call \_correct\_cell\_location

mov dl, byte [ds:gameOfLife\_grid1+bx]

cmp dl, ' '

je \_end\_return

inc ax

\_end\_return:

mov bx, ax

pop dx

pop ax

ret

; helper method for \_check\_adjacent\_cells

; takes a cell location in bx

; returns a corrected cell location in bx

\_correct\_cell\_location:

cmp bx, 0

jl \_less\_than

cmp bx, 199

jg \_greater\_than

jmp \_return

\_less\_than:

add bx, 200

jmp \_return

\_greater\_than:

sub bx, 200

\_return:

ret

; RPN Calculator task

; prints visuals such as expression being entered and result

; processes RPN string when enter is pressed

\_rpnCalculator:

cmp byte [rpn\_evaluate], 1

je rpn\_doEvaluate

jmp rpn\_printString ; skip to printing if no numbers to crunch

rpn\_doEvaluate:

mov si, rpn\_string - 1

; si points to next character

rpn\_expression:

inc si

; check if char is a number

cmp byte [si], '0'

jge rpn\_aboveZero

jmp rpn\_notNumber

rpn\_aboveZero:

cmp byte [si], '9'

jle rpn\_isNumber

jmp rpn\_notNumber

rpn\_isNumber:

cmp byte [rpn\_enteringNum], 1

je rpn\_addToCurrentNum

; if not entering num, now we are

; set flag and clear curNum

mov byte [rpn\_enteringNum], 1

mov word [rpn\_curNum], 0 ; set curNum to 0

rpn\_addToCurrentNum:

mov bx, [si] ; save input to bx

and bx, 0x00FF ; only want lower bits (input is a char)

sub bx, '0' ; convert input from ASCII to int value

mov ax, [rpn\_curNum]

mov cx, 10 ; multiply curNum by 10, then add new char

imul cx

add ax, bx

mov word [rpn\_curNum], ax

jmp rpn\_expression

rpn\_notNumber:

; below logic:

; if entering number, finish number (push to stack)

; then check for operator

; ignore character if not operator

; else perform operation

cmp byte [rpn\_enteringNum], 1

jne rpn\_checkOperator

jmp rpn\_numberDone ; finish entering number

rpn\_checkOperator:

; compare si to rpn\_strPointer

; if equal, print result and exit loop

mov ax, si

sub ax, rpn\_string

cmp ax, [rpn\_strPointer]

jne rpn\_notEnd

jmp rpn\_exprDone

rpn\_notEnd:

; check each operator

; if [si] matches one, do operation

; else ignore char and get input again

cmp byte [si], '+'

je rpn\_addition

jmp rpn\_notPlus

rpn\_addition:

call \_rpn\_pop\_value

; after EVERY push or pop, we have to check DX to see if any errors occurred

cmp dx, 1

jne rpn\_addPop1OK

mov ax, rpn\_underflowStr

jmp rpn\_error

rpn\_addPop1OK:

mov bx, ax ; save first value

call \_rpn\_pop\_value

cmp dx, 1

jne rpn\_addPop2OK

mov ax, rpn\_underflowStr

jmp rpn\_error

rpn\_addPop2OK:

add ax, bx

call \_rpn\_push\_value

cmp dx, 1

jne rpn\_addPushOK

mov ax, rpn\_overflowStr

jmp rpn\_error

rpn\_addPushOK:

jmp rpn\_expression

rpn\_notPlus:

cmp byte [si], '-'

je rpn\_subtraction

jmp rpn\_notMinus

rpn\_subtraction:

call \_rpn\_pop\_value

cmp dx, 1

jne rpn\_subPop1OK

mov ax, rpn\_underflowStr

jmp rpn\_error

rpn\_subPop1OK:

mov bx, ax

call \_rpn\_pop\_value

cmp dx, 1

jne rpn\_subPop2OK

mov ax, rpn\_underflowStr

jmp rpn\_error

rpn\_subPop2OK:

sub ax, bx

call \_rpn\_push\_value

cmp dx, 1

jne rpn\_subPushOK

mov ax, rpn\_overflowStr

jmp rpn\_error

rpn\_subPushOK:

jmp rpn\_expression

rpn\_notMinus:

cmp byte [si], '~'

je rpn\_negation

jmp rpn\_notTilde

rpn\_negation:

call \_rpn\_pop\_value

cmp dx, 1

jne rpn\_negPopOK

mov ax, rpn\_underflowStr

jmp rpn\_error

rpn\_negPopOK:

neg ax

call \_rpn\_push\_value

cmp dx, 1

jne rpn\_negPushOK

mov ax, rpn\_overflowStr

jmp rpn\_error

rpn\_negPushOK:

jmp rpn\_expression

rpn\_notTilde:

cmp byte [si], '\*'

je rpn\_multiplication

jmp rpn\_notAstrisk

rpn\_multiplication:

call \_rpn\_pop\_value

cmp dx, 1

jne rpn\_mulPop1OK

mov ax, rpn\_underflowStr

jmp rpn\_error

rpn\_mulPop1OK:

mov bx, ax

call \_rpn\_pop\_value

cmp dx, 1

jne rpn\_mulPop2OK

mov ax, rpn\_underflowStr

jmp rpn\_error

rpn\_mulPop2OK:

imul bx

call \_rpn\_push\_value

cmp dx, 1

jne rpn\_mulPushOK

mov ax, rpn\_overflowStr

jmp rpn\_error

rpn\_mulPushOK:

jmp rpn\_expression

rpn\_notAstrisk:

cmp byte [si], '/'

je rpn\_division

jmp rpn\_notSlash

rpn\_division:

call \_rpn\_pop\_value

cmp dx, 1

jne rpn\_divPop1OK

mov ax, rpn\_underflowStr

jmp rpn\_error

rpn\_divPop1OK:

mov bx, ax

call \_rpn\_pop\_value

cmp dx, 1

jne rpn\_divPop2OK

mov ax, rpn\_underflowStr

jmp rpn\_error

rpn\_divPop2OK:

cmp bx, 0 ; check for divide by 0

jne rpn\_continueDivide

; divide by 0!

mov ax, rpn\_div0Str

jmp rpn\_error

rpn\_continueDivide:

; check sign of divisor--need to sign extend into DX if negative

cmp ax, 0

jge rpn\_divPositive

mov dx, 0xFFFF

jmp rpn\_divMovDXDone

rpn\_divPositive:

mov dx, 0

rpn\_divMovDXDone:

idiv bx

call \_rpn\_push\_value

cmp dx, 1

jne rpn\_divPushOK

mov ax, rpn\_overflowStr

jmp rpn\_error

rpn\_divPushOK:

jmp rpn\_expression

rpn\_notSlash:

cmp byte [si], '%'

je rpn\_modulus

jmp rpn\_expression ; if we get here, the character isn't an operator, ignore

rpn\_modulus:

call \_rpn\_pop\_value

cmp dx, 1

jne rpn\_modPop1OK

mov ax, rpn\_underflowStr

jmp rpn\_error

rpn\_modPop1OK:

mov bx, ax

call \_rpn\_pop\_value

cmp dx, 1

jne rpn\_modPop2OK

mov ax, rpn\_underflowStr

jmp rpn\_error

rpn\_modPop2OK:

cmp bx, 0 ; check for divide by 0

jne rpn\_continueMod

; divide by 0!

mov ax, rpn\_div0Str

jmp rpn\_error

rpn\_continueMod:

; check sign of divisor--need to sign extend into DX if negative

cmp ax, 0

jge rpn\_modPositive

mov dx, 0xFFFF

jmp rpn\_modMovDXDone

rpn\_modPositive:

mov dx, 0

rpn\_modMovDXDone:

mov dx, 0

idiv bx

mov ax, dx ; remainder stored in dx

call \_rpn\_push\_value

cmp dx, 1

jne rpn\_modPushOK

mov ax, rpn\_overflowStr

jmp rpn\_error

rpn\_modPushOK:

jmp rpn\_expression

rpn\_numberDone:

mov ax, word [rpn\_curNum]

call \_rpn\_push\_value

cmp dx, 1

jne rpn\_curNumPushOK

mov ax, rpn\_overflowStr

jmp rpn\_error

rpn\_curNumPushOK:

; set curNum & enteringNum to 0

mov word [rpn\_curNum], 0

mov byte [rpn\_enteringNum], 0

; if char is 0 (NULL-terminator), done

; else still need to process it

cmp byte [si], 0

je rpn\_exprDone

jmp rpn\_checkOperator

rpn\_exprDone:

; set curNum & enteringNum to 0

mov word [rpn\_curNum], 0

mov byte [rpn\_enteringNum], 0

; print value of expression

call \_rpn\_pop\_value

cmp dx, 1

jne rpn\_popAnsOK

mov ax, rpn\_underflowStr

jmp rpn\_error

rpn\_popAnsOK:

; go digit-by-digit and convert decimal number to string, then print

; clear old result string from '=' to end

push ax

mov ax, rpn\_resultStr

add ax, 4

call \_clearString

pop ax

mov si, rpn\_resultStr

add si, 4 ; gets us past the '='

; first, check if answer is 0 (edge case)

; just put in string and break if so

cmp ax, 0

jne rpn\_ansNonzero

mov byte [si], '0'

jmp rpn\_conversionDone

rpn\_ansNonzero:

mov cx, ax ; to protect the answer, since idiv messes with AX

; we'll divide by BX to get digits

; largest number of digits that can fit in 16 bits is 5

; so start with BX = 10000

mov bx, 10000

; DI will hold a flag to help ignore leading 0s

; not actually using it for indexing, I just need another register

mov di, 0

cmp cx, 0

jge rpn\_decimalToString

; negative answer, print '-' sign and do 2's complement conversion on AX

mov byte [si], '-'

neg cx

inc si

rpn\_decimalToString:

cmp bx, 0

je rpn\_conversionDone

; next digit is ans (in CX) / BX

mov dx, 0

mov ax, cx

idiv bx

; if digit is 0 and we haven't seen anything else (i.e. DI == 0)

; ignore it, it's a leading 0

cmp ax, 0

jne rpn\_addDigit

cmp di, 0

je rpn\_nextDigit

rpn\_addDigit:

mov di, 1

add ax, '0' ; convert to ASCII char

mov byte [si], al ; and add to string

inc si

sub ax, '0' ; get back to the decimal value for next steps

rpn\_nextDigit:

; subtract digit \* BX from CX to ditch the leading digit

imul bx

sub cx, ax

; divide BX by 10

mov dx, 0

mov ax, bx

mov bx, 10

idiv bx

mov bx, ax

jmp rpn\_decimalToString

rpn\_conversionDone:

mov bl, 0

mov bh, 10

mov ch, 2

mov cl, 0

mov dh, 0

mov ax, rpn\_resultStr

call \_printString

jmp rpn\_cleanUp

; before jumping here, put error msg address in AX

rpn\_error:

mov bl, 0

mov bh, 10

mov cl, 0

mov ch, 4

mov dh, 0

call \_printString

rpn\_cleanUp:

; print just-evaluated (or crashed) string below

mov ax, rpn\_string

mov bx, rpn\_lastStr

call \_strcpy

mov ax, bx

mov bl, 0

mov bh, 8

mov cl, 0

mov ch, 8

mov dh, 0

call \_printString

mov word [rpn\_top], 0 ; reset rpn\_top to top of rpn\_stack

call \_clearRPNString ; clear rpn\_string

mov byte [rpn\_evaluate], 0 ; turn off evaluate flag

; and we're done!

rpn\_printString:

mov bl, 0

mov bh, 6

mov cl, 0

mov ch, 7

mov dh, 0

mov ax, rpn\_string

call \_printString

jmp rpn\_end

rpn\_end:

call \_yield

jmp \_rpnCalculator

; helper function for \_rpnCalculator

; pushes number in AX to rpn\_stack

; clobbers DX

; returns 0 in DX if successful, 1 otherwise

\_rpn\_push\_value:

push ax

push bx

push cx

push di

; check for stack overflow

cmp word [rpn\_top], 16

; if rpn\_top == rpn\_stack + 16, stack is full, stack overflow error

jne doPush

mov dx, 1

jmp end\_push\_value

doPush:

mov di, rpn\_stack

add di, [rpn\_top]

add di, [rpn\_top] ; add twice beacuse rpn\_stack contains words (2 bytes)

mov [di], ax

mov dx, 0

inc word [rpn\_top]

end\_push\_value:

pop di

pop cx

pop bx

pop ax

ret

; helper function for \_rpnCalculator

; pops number from rpn\_stack

; clobbers AX, DX

; returns popped value in AX

; 0 in DX if successful, 1 otherwise

\_rpn\_pop\_value:

push bx

push cx

push di

; check for stack underflow

cmp word [rpn\_top], 0

; if rpn\_top == rpn\_stack, stack is empty, stack underflow error

jne doPop

mov dx, 1

jmp end\_pop\_value

doPop:

dec word [rpn\_top]

mov di, rpn\_stack

add di, [rpn\_top]

add di, [rpn\_top] ; add twice beacuse rpn\_stack contains words (2 bytes)

mov ax, [di]

mov dx, 0

end\_pop\_value:

pop di

pop cx

pop bx

ret

; helper function for modifying <rpn\_string>

; takes char in al and appends it to <rpn\_string> at location of <rpn\_strPointer>

; if al is 0, removes last char (replaces it with 0) and decrements <rpn\_strPointer>

; if <rpn\_strPointer> is at beginning/end of the <rpn\_string>, function does nothing if

; remove/append operation is requested

; clobbers nothing

; returns nothing

\_addToRPNString:

push si

push ax

push bx

push cx

push dx

cmp byte [rpn\_evaluate], 1

je end\_addToRPNString

\_notEvaluating:

mov si, rpn\_string

cmp al, 0

jne rpnAppend

jmp rpnBackspace

rpnAppend:

; if <rpn\_strPointer> is pointing one past the end, do nothing

cmp word [rpn\_strPointer], 54

jne doAppend

jmp end\_addToRPNString

doAppend:

add si, [rpn\_strPointer]

mov byte [si], al

inc word [rpn\_strPointer]

; move cursor ahead

mov ah, 0x03

mov bh, 0

int 0x10

inc dl

mov ah, 0x02

int 0x10

jmp end\_addToRPNString

rpnBackspace:

; if <rpn\_strPointer> is pointing to beginning, do nothing

cmp word [rpn\_strPointer], 0

jne doBackspace

jmp end\_addToRPNString

doBackspace:

dec word [rpn\_strPointer]

add si, [rpn\_strPointer]

mov byte [si], " "

; move cursor back

mov ah, 0x03

mov bh, 0

int 0x10

dec dl

mov ah, 0x02

int 0x10

jmp end\_addToRPNString

end\_addToRPNString:

pop dx

pop cx

pop bx

pop ax

pop si

ret

; helper function for clearing <rpn\_string>

; sets all characters of <rpn\_string> to spaces and resets cursor position

; clobbers nothing

; returns nothing

\_clearRPNString:

push si

mov si, rpn\_string

add si, [rpn\_strPointer]

clrrpn\_clearLoop:

cmp si, rpn\_string

jne clrrpn\_clearChar

jmp end\_clearRPNString

clrrpn\_clearChar:

dec si

mov byte [si], " "

jmp clrrpn\_clearLoop

end\_clearRPNString:

mov byte [rpn\_strPointer], 0

; reset cursor

mov ah, 0x02

mov bh, 0

mov dh, 3

mov dl, 0

int 0x10

pop si

ret

; more general function than \_clearRPNString

; sets all characters of string pointed to by AX to spaces

; clobbers nothing

; returns nothing

\_clearString:

push si

mov si, ax

clearLoop:

cmp byte [si], 0 ; stop at NULL-terminator

jne clearChar

jmp end\_clearString

clearChar:

mov byte [si], ' '

inc si

jmp clearLoop

end\_clearString:

pop si

ret

; copies chars from string pointed to by AX into string pointed to by BX

; assumes [AX] string is same length as [BX] string

; clobbers nothing

; returns nothing

\_strcpy:

push si

push di

push dx

mov si, ax

mov di, bx

copyLoop:

cmp byte [si], 0 ; stop at NULL-terminator

jne copyChar

jmp end\_strcpy

copyChar:

mov dl, byte [si]

mov byte [di], dl

inc si

inc di

jmp copyLoop

end\_strcpy:

pop dx

pop di

pop si

ret

; prints a char to the screen using 0x10 interrupt

; bl and bh are thee coordinates of where the char gets printed

; cl is the background color, ch is the foreground color

; dh is whether or not the charachter blinks

; dl is the ascii value of the character to be printed

; video mode must already be set, or else it erases everything

; prints char to row, col stored in BH, BL (respectively)

; clobbers nothing

; returns nothing

\_printChar:

; bx is location, bl is x, bh is y

; cx is color, ch is foreground, cl is background

; dh is blink

; dl is ascii value

push ax

push bx

push cx

push dx

mov ax, 0xB800 ; where the graphics start in memory

mov es, ax

mov al, bh ; do the math to find the character offset

mov ax, 80

mul bh

push dx

xor dx, dx

mov dl, bl

add ax, dx

pop dx

mov bx, ax; offset to move the char to a location on the screen (y \* 80) + x

; bx now holds the right offset

; move blink into position

mov al, dh

shl ax, 3

; move background into position

and cl, 0x7

or al, cl

shl ax, 4

; move foreground into position

and ch, 0xf

or al, ch

shl ax, 8

; move ascii char into position

or al, dl

; move ax into dx, because ax is used for arguments to the video mode

mov dx, ax

mov word [es:bx], dx ; print the character (with formatting) stored in ax in the location stored in bx

pop dx

pop cx

pop bx

pop ax

inc bl ; add one to the x, so the next char can be printed right next to it

inc bl

ret ; return to caller

; print NULL-terminated string to screen=

; takes NULL-terminated string pointed to by DS:AX

; prints to row, col stored in BH, BL (respectively)

; clobbers nothing

; returns nothing

\_printString:

push ax ; save registers

push bx

push cx

push dx

push si

mov si, ax

mov al, bl ; store beginning of line col for new line jumps

.loop:

mov dl, [si] ; DL = current character

inc si ; advance SI to point at next character

cmp dl, 0 ; if (DL == 0), stop

jz .end

cmp dl, 10 ; if newline, jump back to the beginning col

je .new

cmp dl, 13 ; if carriage return, jump down a row

je .ret

jmp .check\_offscreen

.new:

mov bl, al ; jump back to the original col

jmp .loop ; don't print the character

.ret:

inc bh

inc bh ; increment one row

jmp .loop ; don't print the character

.check\_offscreen: ; check if offscreen row or column

cmp bl, 160

jae .loop

cmp bh, 50

jae .end

.print:

call \_printChar ; use \_printChar to print the char

jmp .loop ; repeat

.end:

pop si ; restore registers (de-clobber)

pop dx

pop cx

pop bx

pop ax

ret ; return to caller

; main function; draws headers/borders, monitors keypresses

\_main:

; set video mode

mov ah, 0x0

mov al, 0x3

int 0x10

; print headers/borders for other tasks

mov bl, 0 ; col

mov bh, 0 ; row

mov cl, 7 ; background

mov ch, 4 ; foreground

mov dh, 0 ; blink

mov ax, exit\_header

call \_printString

mov bl, 0

mov bh, 2

mov cl, 7

mov ch, 0

mov dh, 0

mov ax, rpn\_header

call \_printString

mov bl, 108

mov bh, 4

mov ch, 7

mov ax, rpn\_rightBorder

call \_printString

mov bl, 0

mov bh, 26

mov ch, 0

mov ax, gameOfLife\_header

call \_printString

mov bl, 40

mov bh, 30

mov ch, 7

mov ax, gameOfLife\_rightBorder

call \_printString

mov bl, 110

mov bh, 8

mov cl, 0

mov ch, 7

mov ax, music\_ascii\_art

call \_printString

; set cursor position to \_rpnCalculator window

mov ah, 0x02

mov bh, 0

mov dh, 3

mov dl, 0

int 0x10

infiniteLoop\_main:

; check for keypress

cmp byte [currentKey], 0x00

jne checkKey\_leftShiftOn

jmp yield\_Main ; no key pressed

checkKey\_leftShiftOn:

; check for shift to set <shift> boolean

cmp byte [currentKey], 0x2A ; left shift pressed

jne checkKey\_rightShiftOn

mov byte [shift], 1

jmp yield\_Main

checkKey\_rightShiftOn:

cmp byte [currentKey], 0x36 ; right shift pressed

jne checkKey\_leftShiftOff

mov byte [shift], 1

jmp yield\_Main

checkKey\_leftShiftOff:

cmp byte [currentKey], 0xAA ; left shift released

jne checkKey\_rightShiftOff

mov byte [shift], 0

jmp yield\_Main

checkKey\_rightShiftOff:

cmp byte [currentKey], 0xB6 ; right shift released

jne checkKey\_esc

mov byte [shift], 0

jmp yield\_Main

checkKey\_esc:

; if ESC pressed, exit program

cmp byte [currentKey], 0x81

jne checkKey\_space

jmp exit\_program

checkKey\_space:

cmp byte [currentKey], 0x39 ; spacebar

jne checkKey\_backspace

mov al, ' '

call \_addToRPNString

jmp yield\_Main

checkKey\_backspace:

cmp byte [currentKey], 0x0E ; backspace

jne checkKey\_enter

mov al, 0

call \_addToRPNString

jmp yield\_Main

checkKey\_enter:

; if enter pressed, turn on rpn\_evaluate flag

; signals \_rpnCalculator to evaluate the postfix string

cmp byte [currentKey], 0x1C ; enter

jne shiftBranch

mov byte [rpn\_evaluate], 1

jmp yield\_Main

shiftBranch:

; start checking for character keypresses (numbers and operators only)

; check operators first (no pattern)

cmp byte [shift], 1

je shiftOn

jmp shiftOff

shiftOn:

; check keys that would be operators if shift is pressed (shift and = is +, shift and 8 is \*, shift 5 is %)

cmp byte [currentKey], 0x0D ; =, i.e. +

jne checkShiftKey\_8

mov al, '+'

call \_addToRPNString

jmp yield\_Main

checkShiftKey\_8:

cmp byte [currentKey], 0x09 ; 8, i.e. \*

jne checkShiftKey\_5

mov al, '\*'

call \_addToRPNString

jmp yield\_Main

checkShiftKey\_5:

cmp byte [currentKey], 0x06 ; 5, i.e. %

jne checkShiftKey\_backtick

mov al, '%'

call \_addToRPNString

jmp yield\_Main

checkShiftKey\_backtick:

cmp byte [currentKey], 0x29 ; `, i.e. ~

jne doneShiftKey

mov al, '~'

call \_addToRPNString

doneShiftKey:

; no more keys to check with shift on

jmp yield\_Main

shiftOff:

; check for operators without shift

cmp byte [currentKey], 0x0C ; -

jne checkKey\_slash

mov al, '-'

call \_addToRPNString

jmp yield\_Main

checkKey\_slash:

cmp byte [currentKey], 0x35 ; /

jne checkKey\_numpadPlus

mov al, '/'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpadPlus:

cmp byte [currentKey], 0x4E ; numpad +

jne checkKey\_numpadMinus

mov al, '+'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpadMinus:

cmp byte [currentKey], 0x4A ; numpad -

jne checkKey\_numpadStar

mov al, '-'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpadStar:

cmp byte [currentKey], 0x37 ; numpad \*

jne checkKey\_numpad0

mov al, '\*'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpad0:

; check for numbers

; numpad numbers don't follow a pattern, so check those first

cmp byte [E0\_on], 1

je checkKey\_numbers ; skip all the number pad if 0xE0 code is present (don't interpret arrow keys as numbers)

cmp byte [currentKey], 0x52 ; numpad 0

jne checkKey\_numpad1

mov al, '0'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpad1:

cmp byte [currentKey], 0x4F ; numpad 1

jne checkKey\_numpad2

mov al, '1'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpad2:

cmp byte [currentKey], 0x50 ; numpad 2

jne checkKey\_numpad3

mov al, '2'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpad3:

cmp byte [currentKey], 0x51 ; numpad 3

jne checkKey\_numpad4

mov al, '3'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpad4:

cmp byte [currentKey], 0x4B ; numpad 4

jne checkKey\_numpad5

mov al, '4'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpad5:

cmp byte [currentKey], 0x4C ; numpad 5

jne checkKey\_numpad6

mov al, '5'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpad6:

cmp byte [currentKey], 0x4D ; numpad 6

jne checkKey\_numpad7

mov al, '6'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpad7:

cmp byte [currentKey], 0x47 ; numpad 7

jne checkKey\_numpad8

mov al, '7'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpad8:

cmp byte [currentKey], 0x48 ; numpad 8

jne checkKey\_numpad9

mov al, '8'

call \_addToRPNString

jmp yield\_Main

checkKey\_numpad9:

cmp byte [currentKey], 0x49 ; numpad 9

jne checkKey\_numbers

mov al, '9'

call \_addToRPNString

jmp yield\_Main

checkKey\_numbers:

; check non-numpad numbers (these follow a pattern)

; numbers run from 0x02 to 0x0B, so we can check the range and get char by its difference from 0x02

cmp byte [currentKey], 0x02

jl yield\_Main ; safe to bail at this point because we've checked all of the cases outside the range

cmp byte [currentKey], 0x0B

jg yield\_Main

jl notZero

; zero comes after other numbers, not before, so we have to handle it separately

mov al, '0'

call \_addToRPNString

jmp yield\_Main

notZero:

mov al, '0'

dec al

add al, [currentKey]

call \_addToRPNString

yield\_Main:

mov byte [currentKey], 0

mov byte [E0\_on], 0 ; turn off 0xE0 flag

call \_yield

; pause after drawing updates

mov ah, 0x86

mov cx, 0

mov dx, 0xFFFF

int 0x15

jmp infiniteLoop\_main

exit\_program:

mov byte [currentKey], 0

; set video mode once more to clear screen

mov ah, 0x0

mov al, 0x3

int 0x10

jmp terminate

; custom timer hardware interrupt

timer:

push ax

mov ax, word [isNotePlaying]

cmp ax, 0

jne \_notePlaying

jmp \_return\_timer

\_notePlaying:

mov ax, [noteDuration]

dec ax

mov [noteDuration], ax

cmp ax, 0

jne \_return\_timer

call \_stopNote

\_return\_timer:

pop ax

; jump to the original INT 8 handler

jmp far [cs:previous8] ; Use CS as the segment here, since who knows what DS is now

; custom keyboard hardware interrupt

keyboard:

push ax

in al, 0x60

cmp al, 0xE0 ; 0xE0 make code?

jne noE0

mov byte [E0\_on], 1

noE0:

mov byte [currentKey], al

mov al, 0x20

out 0x20, al

pop ax

iret

SECTION .data

; global variables

; strings

exit\_header: db " -- Press ESC to exit -- ", 0

rpn\_header: db " RPN Calculator Music ", 0

rpn\_rightBorder: times 11 db " ", 13, 10

db 0

; Game of Life strings

gameOfLife\_header: db " John Conway's ", 13, 10

db " Game of Life Graphics ", 0

; grid with a 'glider'

gameOfLife\_grid1: db ' \* \* \*\*\* '

gameOfLife\_grid2: db ' '

gameOfLife\_rightBorder: times 10 db " ", 13, 10

db 0

; tasks A & B

taskA\_str: db "I am task A", 0

taskB\_str: db "I am", 13, 10, "task B", 0

taskA\_dir: db 1

taskB\_dir: db 0

; ball graphics

ball\_dirX: db 1

ball\_dirY: db 0

; RPN calculator

rpn\_string: times 54 db " " ; max length: 54

db 0

rpn\_strPointer: dw 0 ; initialized to 0

rpn\_stack: times 16 dw 0

rpn\_top: dw 0

rpn\_curNum: dw 0

rpn\_enteringNum: db 0 ; bool variable to track if last input was a number

rpn\_evaluate: db 0

rpn\_lastStr: times 54 db " "

db 0

rpn\_resultStr: db " = ", 0

rpn\_underflowStr: db " Stack underflow! ", 0

rpn\_overflowStr: db " Stack overflow! ", 0

rpn\_div0Str: db " Divide by 0! ", 0

; music task

; ascii art taken from https://www.ascii-code.com/ascii-art/music/musical-notation.php

music\_ascii\_art: db " |\\ ", 13, 10, "----|\----|-\\--- |\\----", 13, 10, "----|/---0---\|---|-\\---", 13, 10, "---/|---------|--0---\|--", 13, 10, "--|-/-\------0--------|--", 13, 10, "---\|/---------------0---", 13, 10, " d", 0

isNotePlaying: db 0

noteDuration: dd 0

song\_notes\_pitch: dw 4063, 3619, 3416, 2711, 3619, 2031, 2280, 2415, 3416, 2711, 2415, 2280, 2031, 2711, 4063, 3043, 3416, 4063, 5423, 0, 4831, 4560, 4063, 3043, 3416, 3043, 2711, 3416, 3619, 3416, 3043, 3619, 4831, 3043, 3416, 5423, 3619, 4560, 4063, 5423, 3619, 4063

song\_notes\_duration: db 9, 9, 9, 9, 35, 9, 9, 9, 9, 35, 9, 9, 9, 17, 17, 9, 9, 9, 17, 17, 17, 17, 17, 17, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 17, 17, 17, 9, 9, 35

note\_pointer: dd 0

; custom keyboard interrupt

; current key scan code

currentKey: db 0

; shift pressed boolean

shift: db 0

; boolean flagging if 0xE0 make code is currently modifying <currentKey>

E0\_on: db 0

; address of previous 0x09 interrupt

previous9: dd 0

; address of previous 0x08 interrupt

previous8: dd 0

; global variables for stacks

current\_task: db 0

stacks: times (256 \* 8) db 0 ; 8 fake stacks of size 256 bytes

task\_status: times 8 db 0 ; 0 means inactive, 1 means active

stack\_pointers: dw 0 ; the first pointer needs to be to the real stack !

dw stacks + (256 \* 1)

dw stacks + (256 \* 2)

dw stacks + (256 \* 3)

dw stacks + (256 \* 4)

dw stacks + (256 \* 5)

dw stacks + (256 \* 6)

dw stacks + (256 \* 7)

dw stacks + (256 \* 8)

## boot.asm

; Bootloader that loads/runs the kernel

; program from the boot disk.

bits 16

section .text

; The BIOS will load us into memory at 0000:7C00h; NASM needs

; to know this so it can generate correct absolute data references.

org 0x7C00

; jump over initial data and start executing code

start: jmp main

; .data "section" (although it's part of .text)

boot\_msg db "CpS 230 Team Project Kernel", 13, 10

db "by Zachary Hayes and Ryan Longacre", 13, 10

db "----------------------------------", 13, 10

db "Press any key to boot kernel", 0

boot\_disk db 0 ; Variable to store the number of the disk we boot from

retry\_msg db "Error reading payload from disk; retrying...", 13, 10, 0

main:

; Set DS == CS (so data addressing is normal/easy)

mov bx, cs

mov ds, bx

; Save the boot disk number (we get it in register DL)

mov byte [boot\_disk], dl

; Set SS == 0x0800 (which will be the segment we load everything into later)

mov bx, 0x0800

mov ss, bx

; Set SP == 0x0000 (stack pointer starts at the TOP of segment; first push decrements by 2, to 0xFFFE)

mov bx, 0x0000

mov sp, bx

; Print the boot message/banner

mov dx, boot\_msg

call puts

; Wait for keypress, then read boot disk

mov ah, 0x00

int 0x16

mov cl, 2

jmp read\_disk

read\_error:

mov dx, retry\_msg

call puts

read\_disk:

; use BIOS raw disk I/O to load 13 sectors (starting at 2) from disk number <boot\_disk> into memory at 0800:0000h (retry on failure)

mov ah, 0x2

mov al, 13

mov ch, 0

mov cl, 2

mov dh, 0

mov dl, [boot\_disk]

mov bx, 0x0800

mov es, bx

mov bx, 0x0000

int 0x13

jc read\_error

; Finally, jump to address 0800h:0000h (sets CS == 0x0800 and IP == 0x0000)

jmp 0x0800:0x0000

; print NULL-terminated string from DS:DX to screen using BIOS (INT 10h)

; takes NULL-terminated string pointed to by DS:DX

; clobbers nothing

; returns nothing

puts:

push ax

push cx

push si

mov ah, 0x0e

mov cx, 1 ; no repetition of chars

mov si, dx

.loop: mov al, [si]

inc si

cmp al, 0

jz .end

int 0x10

jmp .loop

.end:

pop si

pop cx

pop ax

ret

; make sure the boot sector signature starts 510 bytes from our origin

times 510 - ($ - $$) db 0

; BOOT SECTOR SIGNATURE (\*must\* be the last 2 bytes of the 512 byte boot sector)

dw 0xaa55