;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; ;

; QUEUE ;

; Queue Routine Include Definitions ;

; EE/CS 51 ;

; Archan Luhar ;

; TA: Joe Greef ;

; ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Defines queue metadata byte offset constants and test function constants

QUEUE\_TEST\_LENGTH EQU 256

; Number of bytes of metadata before the queue elements data starts

QUEUE\_QUEUE\_OFFSET EQU 7

; Maximum number of bytes of queue data in the queue struct.

; The queue struct must me initialized with less bytes than this max number.

; E.g. 256 word sized elements = 512 bytes: good

; E.g. 512 byte sized elements = 512 bytes: good

; E.g. 512 word sized elements = 1024 bytes: good

; E.g. 1024 word sized elements = 2048 bytes: not good

QUEUE\_MAX\_BYTES EQU 1024

; Defines the number of bytes in the two possible element sizes byte and word

ELEM\_BYTE\_SIZE EQU 1

ELEM\_WORD\_SIZE EQU 2

; Queue Structure which defines the metadata and the start of the queue

; elements data.

; elem\_size: 1 if each element is byte, 2 if each element is a word

; len: Max number of elements in queue

; head\_index: Number of elements offset from start of queue elements data

; count: Current number of elements in the queue.

; queue: Start of queue elements.

queueSTRUC STRUC

elem\_size DB ?

len DW ?

head\_index DW ?

count DW ?

queue DB QUEUE\_MAX\_BYTES DUP (?)

queueSTRUC ENDS

NAME HW3MAIN

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; ;

; HW3MAIN ;

; Homework 3 Main Loop ;

; EE/CS 51 ;

; Archan Luhar ;

; TA: Joe Greef ;

; ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Description: This program allocates space for myQueue and calls my own

; defined MyQueueTest which is used for stepping through

; element enqueuing and dequeuing to check correct memory.

; This program also calls the QueueTest function provided by

; Glen.

;

; Input: None.

; Output: None.

;

; User Interface: None. User can set breakpoint at MyQueueTest to step through

; sample queue additions and removals to see changes in

; the memory storing the queue.

; If QueueTest succeeds, infinite loop occurs at

; breakpoint hw3test.QueueGood.

;

; Error Handling: If QueueTest fails, infinite loop occurs at breakpoint

; hw3test.QueueError.

;

; Algorithms: None.

; Data Structures: Queue struct is defined in queue.inc. It uses a cyclic array

;

; Known Bugs: None.

; Limitations: There must be less than 1024 bytes of elements.

;

; Revision History:

; 11/02/13 Archan Luhar Created hw3main.asm. Contains main function

; that calls test functions. Also allocates

; queue struct in DS.

; Include file defines queue metadata offset constants

$INCLUDE(queue.inc)

CGROUP GROUP CODE

DGROUP GROUP DATA, STACK

CODE SEGMENT PUBLIC 'CODE'

ASSUME CS:CGROUP, DS:DGROUP

;external function declarations

EXTRN QueueInit:NEAR

EXTRN QueueEmpty:NEAR

EXTRN QueueFull:NEAR

EXTRN Dequeue:NEAR

EXTRN Enqueue:NEAR

EXTRN QueueTest:NEAR

START:

MAIN:

MOV AX, DGROUP ;initialize the stack pointer

MOV SS, AX

MOV SP, OFFSET(DGROUP:TopOfStack)

MOV AX, DGROUP ;initialize the data segment

MOV DS, AX

MOV SI, OFFSET(myQueue) ; Let SI be the pointer to the queue

MOV AX, QUEUE\_TEST\_LENGTH ; Set size to that defined in inc

MOV BL, ELEM\_BYTE\_SIZE ; Set element size to byte

CALL QueueInit ; Initialize the queue

CALL MyQueueTest ; Test out the queue briefly

MOV CX, QUEUE\_TEST\_LENGTH ; Pass the queue and queue length to

CALL QueueTest ; provided test function.

; Enqueues a bunch of numbers to see if properly stored. Must use debugger

; to verify queue data in memory at SI.

MyQueueTest:

MOV AL, 1002H;

CALL Enqueue;

MOV AX, 3004H;

CALL Enqueue;

MOV AX, 5006H;

CALL Enqueue;

MOV AX, 7008H;

CALL Enqueue;

MOV AX, 9000H;

CALL Enqueue;

CALL QueueEmpty;

CALL QueueFull;

CALL Dequeue;

DATA SEGMENT PUBLIC 'DATA'

myQueue queueSTRUC <>

DATA ENDS

STACK SEGMENT STACK 'STACK'

DB 80 DUP ('Stack ') ;240 words

TopOfStack LABEL WORD

STACK ENDS

CODE ENDS

END START

NAME QUEUE

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; ;

; QUEUE ;

; Queue Routines ;

; EE/CS 51 ;

; Archan Luhar ;

; TA: Joe Greef ;

; ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Description: This file contains several routines to manipulate and read

; data from a queues structure in memory:

; QueueInit, QueueEmpty, QueueFull, Dequeue, Enqueue

;

; Input: None.

; Output: None.

;

; User Interface: None.

;

; Error Handling: Enqueing and dequeuing block until queue is valid.

;

; Algorithms: None.

; Data Structures: Queue struct is defined in queue.inc. It uses a cyclic array

;

; Known Bugs: None.

; Limitations: There must be less than 1024 bytes of elements.

;

; Revision History:

; 10/28/13 Archan Luhar Initial outline.

; 11/02/13 Archan Luhar Finished HW2. Passes tests.

; Include file defines queue struct and offset constants

$INCLUDE(queue.inc)

CGROUP GROUP CODE

CODE SEGMENT PUBLIC 'CODE'

ASSUME CS:CGROUP

; QueueInit

;

; Description: This function is used to create a queue of a given length

; and given element size at a given address.

;

; Operation: This function writes the meta data of the queue in the first

; byte and three words of the queue: the size of each element,

; the max number of elements, the index of the head (0), and

; the count of elements in the queue also initialized to 0.

; The start of the queue elements would be the eigth byte.

;

; Arguments: AX - the length, max number of elements in the queue.

; SI - the location at which to initialize the the queue.

; BL - size of each element (0: bytes, 1: words)

;

; Return Value: None.

;

; Local Variables: None.

;

; Shared Variables: None.

; Global Variables: None.

;

; Input: None.

; Output: None.

;

; Error Handling: None.

;

; Algorithms: None.

;

; Data Structures: Cyclic array

;

; Registers Used: AX (return value)

;

; Stack Depth: 0

;

; Author: Archan Luhar

; Last Modified: 11/02/2013

;

;

; Pseudo Code

; -----------

; queue.elem\_size = size ? 2 : 1 ; queue's size: word if nonzero, byte if 0

; queue.len = len ; set queue's length

; queue.head\_index = 0 ; set queue's head index

; queue.count = 0 ; set queue's count of number of elements

;

; queueSize = len \* queue.elem\_size

QueueInit PROC NEAR

PUBLIC QueueInit

InitQueueInit:

CMP BL, 0 ; Check the argument size

JE SetQueueSizeByte ; If zero, then set to byte size element

SetQueueSizeWord:

MOV [SI].elem\_size, ELEM\_WORD\_SIZE ; If non-zero, element size is word.

JMP SetQueueLength ; Jump over setting size to byte.

SetQueueSizeByte:

MOV [SI].elem\_size, ELEM\_BYTE\_SIZE

; JMP SetQueueLength;

SetQueueLength:

MOV [SI].len, AX ; Set the number of elements from AX argument

SetQueueHeadAndCount:

MOV [SI].head\_index, 0 ; Initialize head index to 0

MOV [SI].count, 0 ; Initialize as empty queue having count 0 elems

EndQueueInit:

RET

QueueInit ENDP

; QueueEmpty

;

; Description: This function is used to see if a given queue is empty.

;

; Operation: This function simply looks at the word five bytes into

; the metadata which stores the count of elements in queue.

; Then it returns true if it is zero, else it returns false.

;

; Arguments: SI - the address of the queue.

;

; Return Value: ZF - 1 if empty, else 0.

;

; Local Variables: None.

;

; Shared Variables: None.

; Global Variables: None.

;

; Input: None.

; Output: None.

;

; Error Handling: None.

;

; Algorithms: None.

;

; Data Structures: Cyclic array

;

; Registers Used: ZF

;

; Stack Depth: 0

;

; Author: Archan Luhar

; Last Modified: 10/28/2013

;

;

; Pseudo Code

; -----------

; return count == 0

QueueEmpty PROC NEAR

PUBLIC QueueEmpty

CMP [SI].count, 0 ; If the number of elements (count) is zero

RET ; the queue is empty. ZF gets set since 0-0 = 0.

QueueEmpty ENDP

; QueueFull

;

; Description: This function is used to see if a given queue is full.

;

; Operation: This function simply looks at the word five bytes into

; the metadata. This word stores the num of elements in queue.

; If it equals the word stored at 1 byte into the metadata,

; the length of the queue, then it returns true, else false.

;

; Arguments: SI - the address of the queue.

;

; Return Value: ZF - 1 if full, else 0.

;

; Local Variables: None.

;

; Shared Variables: None.

; Global Variables: None.

;

; Input: None.

; Output: None.

;

; Error Handling: None.

;

; Algorithms: None.

;

; Data Structures: Cyclic array

;

; Registers Used: ZF

;

; Stack Depth: 0

;

; Author: Archan Luhar

; Last Modified: 11/02/2013

;

;

; Pseudo Code

; -----------

; return queue.count == queue.length

QueueFull PROC NEAR

PUBLIC QueueFull

PUSH BX

MOV BX, [SI].len ; BX contains the length of the queue

CMP [SI].count, BX ; If the count == the length, the queue is full.

POP BX ; ZF gets set if full since count-len = 0.

RET

QueueFull ENDP

; Dequeue

;

; Description: This function returns the value at the head of the queue.

; It is a blocking function that waits until there is a value

; if initially the queue is empty.

;

; Operation: This function loops, waiting, until the queue is not empty.

; Then, it stores the head in AL if element size is byte.

; Else, element size is word so it stores the head in AX.

; It then decrements the count.

; And also it sets the head to (head + 1) mod (length - 1).

; The location to read the value would be

;

; Arguments: SI - the address of the queue.

;

; Return Value: AX if element size is word, else AL - the head of queue.

;

; Local Variables: None.

;

; Shared Variables: None.

; Global Variables: None.

;

; Input: None.

; Output: None.

;

; Error Handling: None.

;

; Algorithms: None.

;

; Data Structures: Cyclic array

;

; Registers Used: AX if element size is word, else AL.

;

; Stack Depth: 0

;

; Author: Archan Luhar

; Last Modified: 11/02/2013

;

;

; Pseudo Code

; -----------

; while (QueueEmpty()): ; block while queue is empty

; continue loop

;

; returnVal = queue.queue[queue.head\_index \* queue.elem\_size]

; queue.headIndex = (queue.headIndex + 1) mod (queue.len)

; queue.count--

; return returnVal

Dequeue PROC NEAR

PUBLIC Dequeue

BlockingDequeue: ; Loops until queue is not empty.

CALL QueueEmpty ; See if queue is empty

JZ BlockingDequeue ; If zero flag is set, it is empty, block.

; JNZ QueueNotEmpty

QueueNotEmpty:

PUSH SI ; Save queue pointer.

PUSH AX ; Save AX since we will use it to store the

; computed offset for the head element.

XOR AX, AX ; Start with offset AX = 0

MOV AL, [SI].elem\_size ; AX = size of each element

PUSH DX ; Save DX in case MUL overflows

MUL [SI].head\_index ; AX = offset from start of queue elems

; = size \* head\_index

POP DX ; Restore DX

ADD AX, QUEUE\_QUEUE\_OFFSET ; AX = size \* head\_index + start of queue offset

; = offset from start of queue pointer

CMP [SI].elem\_size, ELEM\_BYTE\_SIZE ; If elem size is byte

JE GetQueueByte ; Then dequeu a byte, else dequeue a word.

GetQueueWord:

ADD SI, AX ; SI = queue ptr SI + offset

POP AX ; Restore AX which we were using for offset

MOV AX, WORD PTR [SI] ; Return value AX contains word element at head

JMP HeadAhead ; Move the head forward to next element

GetQueueByte:

ADD SI, AX ; SI = queue ptr SI + offset

POP AX ; Restore AX which we were using for offset

MOV AL, BYTE PTR [SI] ; Return value AL contains byte element at head

; JMP HeadAhead ; Move the head forward to next element

HeadAhead:

POP SI ; SI = queue ptr

PUSH AX ; Save return value.

MOV AX, [SI].head\_index ; Computing next head index in AX = head\_index

INC AX ; Increment head index

PUSH BX ; Save BX

MOV BX, [SI].len ; BX = max number of elements in queue

PUSH DX ; Save DX

MOV DX, 0 ; Setup DX for division

DIV BX ; AX = head index / len. DX = head index mod len

MOV AX, DX ; If AX > len - 1, wrap around to 0 since

POP DX ; DX contains remainder. Return DX to original.

POP BX ; Return BX to original..

MOV [SI].head\_index, AX ; Save the new head index back into queue data

POP AX ; Return AX back to dequeued elem return value

EndDequeue:

DEC [SI].count ; Since we've dequeued, decrement count

RET

Dequeue ENDP

; Enqueue

;

; Description: This function pushes to the end of a given queue a given

; value.

; It is a blocking function that waits until the queue is

; not full to enqueue the value.

;

; Operation: This function loops, waiting, until the queue is not full.

; Then it increments the count.

; The tail index is just (head index + count) mod (length - 1)

; If element size is byte, it stores argument from AL at tail.

; Elese element size is word so it stores argument from AX

; at tail.

; The location to store would be start of queue elements +

; tail index \* element size.

;

; Arguments: SI - the address of the queue.

; AX if element size is word, else AL - value to enqueue

;

; Return Value: None.

;

; Local Variables: None.

;

; Shared Variables: None.

; Global Variables: None.

;

; Input: None.

; Output: None.

;

; Error Handling: None.

;

; Algorithms: None.

;

; Data Structures: Cyclic array

;

; Registers Used: None.

;

; Stack Depth: 0

;

; Author: Archan Luhar

; Last Modified: 11/02/2013

;

;

; Pseudo Code

; -----------

; while (QueueFull()): ; block while queue is full

; continue loop

; queue.count++

; tailIndex = (queue.headIndex + queue.count) mod (queue.length)

; queue.queue[tailIndex \* queue.elem\_size] = value

Enqueue PROC NEAR

PUBLIC Enqueue

BlockingEnqueue: ; Block until queue is not full.

CALL QueueFull ; Sets zero flag if full

JZ BlockingEnqueue ; If zero flag is set, loop.

; JNZ QueueNotFull

QueueNotFull:

PUSH SI ; Save SI queue ptr

PUSH AX ; Save argument enqueue value

MOV AX, [SI].head\_index ; AX = head index

ADD AX, [SI].count ; AX = head index + count

PUSH BX ; Save BX to use for len

MOV BX, [SI].len ; BX = len

PUSH DX ; Save DX

MOV DX, 0 ; Setup DX for division

DIV BX ; AX = (head index + count) / len

MOV AX, DX ; AX = DX = (head index + count) mod len

POP DX ; Restore DX

; AX now contains tail index.

POP BX ; Restore BX

; multiply index by size

PUSH DX ; Save DX incase multiplication overflow

MUL [SI].elem\_size ; AX = tail offset from start of queue elems

POP DX ; Restore DX

ADD AX, QUEUE\_QUEUE\_OFFSET ; AX = tail offset from start of queue ptr

CMP [SI].elem\_size, ELEM\_BYTE\_SIZE ; If elem size is byte,

JE SetQueueByte ; Write byte to queue, else write word.

SetQueueWord:

ADD SI, AX ; SI = SI queue ptr + tail offset

POP AX ; Restore enqueue value argument

MOV WORD PTR [SI], AX ; Write enqueue word value argument to tail

JMP EndEnqueue ; Jump over writing a byte to tail

SetQueueByte:

ADD SI, AX ; SI = SI queue ptr + tail offset

POP AX ; Restore enqueue value argument

MOV BYTE PTR [SI], AL ; Write enqueue byte value argument to tail

; JMP EndEnqueue

EndEnqueue:

POP SI ; Restore original queue ptr

INC [SI].count ; Increment count of number of elems in queue

RET

Enqueue ENDP

CODE ENDS

END

# EE/CS 51

# HW3 - Queue Routines

# Archan Luhar

# TA: Joe Greef

# Makefile.mak

all: assemble link locate

check:

asm86chk queue.asm

asm86chk hw3main.asm

assemble:

asm86 queue.asm m1 ep db

asm86 hw3main.asm m1 ep db

link:

link86 hw3main.obj,queue.obj,hw3test.obj

locate:

loc86 hw3main.lnk