HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION TECHNOLOGY AND COMMUNICATION
——— o0o ———

Computer Architecture Lab Report

Instructors : Prof. Le Ba Vui

Group 1 : Nguyen Bui Duc Anh - 20220070
Vuong Le Binh Duong - 20226035

Hanoi, December 2024

Contents

1 Memory Allocation using malloc()

The following program involves the implementation of a dynamic memory allocation function (`malloc`) with examples written in RISC-V assembly language. This exercise focuses on fixing memory alignment issues and creating utility functions to manage memory.

Tasks

1. **Fix Memory Alignment Error:** Ensure that word memory allocation adheres to the rule that the word address must be divisible by 4.
2. **Retrieve Pointer Value:** Write a function to fetch the value stored at a pointer.
3. **Retrieve Pointer Address:** Create a function to return the address of a pointer.
4. **Copy Two Pointers of Character:** Implement a function to copy the values between two character pointers.
5. **Free Allocated Memory:** Write a function to deallocate memory pointed to by a pointer.
6. **Calculate Allocated Memory:** Design a function to calculate and return the total amount of allocated memory.
7. **2D Array Allocation using malloc2:** Develop a function to allocate a 2-dimensional array of type `.word`. The function should take the following parameters:
   - The starting address of the array
   - The number of rows
   - The number of columns
8. **Access and Update 2D Array Values:** Based on Task 7, implement the following functions:
   - `getArray[i][j]`: Retrieve the value at row i and column j.
   - `setArray[i][j]`: Set a new value at row i and column j.

1.1 Background

The `malloc` function, short for memory allocation, is a standard library function in the C programming language used for dynamic memory management. It is part of the stdlib.h librabry and provides a mechanism to allocate memory during runtime rather than compile-time. Dynamic memory allocation is essential in applications where memory requirements are unknown beforehand, such as programs involving arrays, linked lists, and complex data structures. Prior to `malloc`, traditional methods of memory allocation (like static arrays) were restrictive because memory sizes had to be predetermined. The introduction of `malloc` enabled programmers to allocate and free memory dynamically, improving memory utilization and program flexibility.

1.2 Project Objectives

The objectives of this project are as follows:

1. Implement functions for managing pointers, including retrieving values, addresses, and copying data.
2. Develop functions to allocate and deallocate memory efficiently.
3. Extend `malloc` functionality to support 2D array allocation and operations such as accessing and updating array elements.
4. Demonstrate a clear understanding of dynamic memory management concepts in RISC-V assembly.

1.3 Project Description

This project implements a simplified version of the malloc function in assembly language, simulating dynamic memory allocation commonly used in higher-level programming languages like C. The program manages a contiguous block of memory (a simulated heap) and performs key operations such as memory allocation, alignment, and pointer management. The brief description about this project includes:

- **Main Menu:** At the start of the program, users are presented with a menu to choose one of the following options:
    1. **Input a character array:** Initialize a `char` array (1-byte units)
    2. **Input a byte array:** Initialize a `byte` array (1-byte units).
    3. **Input a word array:** Initialize a `word` array (4-byte units).
    4. **Input a 2D word array:** Initialize a 2-dimensional `word` array, specifying rows and columns.

5. **Display pointer addresses and current values:** Show the memory addresses and the current values stored by the used pointers.
6. **Free allocated memory:** Release the memory that was dynamically allocated to avoid memory leaks.
7. **Calculate allocated memory size:** Compute the total amount of memory (byte) that has been allocated so far.
8. **Copy character pointers and values:** Check conditions and copy the values of input characters to the defined char array which the first element pointer has been stored.
9. **Exit:** Terminate the program.

- **Dynamic Memory Management:**
  - `Sys_MyFreeSpace` is initialized to create free space for dynamic memory allocation.
- **Malloc Working Mechanism:**
  - The `malloc` function in this assembly code dynamically allocates memory by managing a simulated heap. It calculates the total memory size to allocate using product of number of elements and size per element.
  - The aligned memory address is stored and returned, and the free memory pointer `Sys_TheTopOfFree` is updated to reflect the new top of available memory.
- **Array Initialization and Input:**
  - Once the user selects an option for initializing an array, the program prompts for:
    - Array length (number of elements).
    - Values of each elements.
- **Pointer Initialization and Storage:**
  - The free memory pointer, `Sys_TheTopOfFree`, is maintained to store the address of the next available memory block for allocation.
  - Each elements will be stored in data segment (the dynamic memory segment) after input process.
  - The pointer to the beginning of the array will be stored in each defined block.
  - After the input process, calculate the total memory allocated then calculate and store the next available memory block to `Sys_TheTopOfFree`.

1.4 Project Functionality

The functionality of this `malloc` implementation is to allocate dynamic memory in a simulated heap and update the free memory pointer. It ensures proper alignment and calculates the total memory needed for the requested allocation:

- **Data Segment:**

```
.data
# String messages
ValueChar:   .asciz "\nChar Pointer Value: "
ValueByte:   .asciz "\nByte Pointer Value: "
ValueWord:   .asciz "\nWord Pointer Value: "
Value2DWord: .asciz "\n2D Word Pointer Value: "
AddressChar: .asciz "Char Pointer Address: "
AddressByte: .asciz "Byte Pointer Address: "
AddressWord: .asciz "Word Pointer Address: "
Address2DWord: .asciz "2D Word Pointer Address: "

enteri: .asciz "Enter i: "
enterj: .asciz "Enter j: "

mainmenu: .asciz "\n----- Menu -----"
newline:   .asciz "\n"
entersize: .asciz "Enter number of index: "
enterelements: .asciz "Enter value: "
message1:   .asciz "\n1. Char\n"
message2:   .asciz "2. Byte\n"
message3:   .asciz "3. Word\n"
message4: .asciz "4. 2D Word\n"
message5: .asciz "5. Display value and address of pointers. \n"
message6: .asciz "6. Free memory allocated. \n"
message7: .asciz "7. Calculate memory allocated. \n"
message8: .asciz "8. Copy char pointer. \n"
message9: .asciz "9. Exit. \n"
errormessage: .asciz "Unvalid input, please enter value from 1 to 9.\n"
prompt:   .asciz "Choose an option: "
notinitialize: .asciz "\nNot Initialize.\n"
memory: .asciz "Number of bytes(memory) allocated: "
return: .asciz "4. Return.\n"

message4.1: .asciz "\n1. Get a[i][j]. \n"
message4.2: .asciz "2. Set a[i][j]. \n"
```

```
message4.3: .asciz "3. Return.\n"
message4.4: .asciz "Value: "
message4.5: .asciz "The value has been update. \n"

message8.1: .asciz "Cannot copy! \n"
message8.2: .asciz "Copy finish. \n"
# Pointer variables
CharPtr:   .word  0 # Character pointer
BytePtr:   .word  0 # Byte pointer
WordPtr:   .word  0 # Word pointer
TwoDWordPtr: .word 0
# System memory management variables
Sys_TheTopOfFree: .word 0
Sys_MyFreeSpace:  .space 10000  # Allocate more space for dynamic memory

CopyPtr: .word 0
```

- **System Initialize Memory Function:**

```
# Initialize memory management system
SysInitMem:
la t0, Sys_TheTopOfFree
la t1, Sys_MyFreeSpace
sw t1, (t0)
ret
```

- **Malloc and Malloc2 Function:**

```
# Dynamic memory allocation
malloc:
    la t0, Sys_TheTopOfFree
    lw t1, 0(t0)    # Load current free memory pointer
    li a5, 3
    bne a6, a5, initialize
    andi t3, t1, 0x03
    beq t3, zero, initialize
    addi t1, t1, 4
    sub t1, t1, t3
```

```
initialize:
    sw t1, (a0)    # Store allocated address
    mv a0, t1     # Return allocated address

    mul t2, a1, a2   # Calculate total size
    add t3, t1, t2  # Update free memory pointer
    sw t3, (t0)
    jr ra

malloc2:
    addi sp, sp, -12     # Save necessary registers on stack
    sw ra, 8(sp)      # Save return address
    sw a1, 4(sp)     # Save a1
    sw a2, 0(sp)      # Save a2

    mul a1, a1, a2      # a1 = number of elements (rows * cols)
    li a2, 4      # a2 = size of one element (word = 4 bytes)
    jal malloc      # Call malloc to allocate memory

    lw ra, 8(sp)     # Restore return address
    lw a1, 4(sp)     # Restore a1
    lw a2, 0(sp)    # Restore a2
    addi sp, sp, 12       # Restore stack pointer
    ret        # Return
```

- **Input and Display Function: (Similarly between Char, Byte, Word and 2D Word)**

```
# Input data for CharPtr
NhapDulieuChar:
    li s3, 0
    la t1, CharPtr    # Load address of CharPtr
    lw t1, (t1)    # Load allocated memory address

NhapDulieuChar_Loop:
    li a7, 4
    la a0, enterelements  # Print message
    ecall
    li a7, 12     # Syscall to read a char
    ecall
    sb a0, 0(t1)      # Store the input char
    addi t1, t1, 1      # Move to next byte
```

```
        addi s3, s3, 1

        li a7, 4
        la a0, newline
        ecall

        bne s3, a1, NhapDulieuChar_Loop
        ret

    DisplayAddressValueChar:
        # Display CharPtr value and address
        la   a0, CharPtr
        lw   t1, 0(a0)         # Load the pointer address
        beq t1, zero, NotInitialize
        lb   t2, 0(t1)        # Load the value pointed by CharPtr

        # Print CharPtr value label
        li   a7, 4
        la   a0, ValueChar
        ecall
        # Print CharPtr value
        li   a7, 11           # Print a character
        mv   a0, t2
        ecall
        # Print newline
        li   a7, 4
        la   a0, newline
        ecall

        # Print CharPtr address label
        li   a7, 4
        la   a0, AddressChar
        ecall
        # Print CharPtr address
        li   a7, 34
        mv   a0, t1
        ecall
        # Print newline
        li   a7, 4
        la   a0, newline
        ecall
    jr ra
```

- **Get and Set in 2D Array:**

```
getArray:
    mul t0, s0, a2    # Element position = i * cols
    add t0, t0, s1    # Add j to get the final position
    slli t0, t0, 2      # Multiply by 4 (word size) to get the byte offset
    add t0, t0, a0     # Add base address to get the actual address
    lw a0, 0(t0)      # Load the value of the element
    ret      # Return
setArray:
    mul t0, s0, a2     # Element position = i * cols
    add t0, t0, s1     # Add j to get the final position
    slli t0, t0, 2     # Multiply by 4 (word size) to get the byte offset
    add t0, t0, a0    # Add base address to get the actual address
    sw s2, 0(t0)    # Store the value into the element
    ret    # Return
```

- **Memory Freeing Function:**

```
Free:
    la   t0, Sys_TheTopOfFree     # Load address of Sys_TheTopOfFree
    lw   t1, 0(t0)        # Load the current top of free memory address

    # Load the beginning of the allocated space
    la   t2, Sys_MyFreeSpace     # This is where the allocated space starts

    # If Sys_TheTopOfFree is equal to or less than the start, nothing to clean
    blt  t1, t2, Free_End

    sw t2, 0(t0)

    # Reset the CharPtr to zero
    la   t0, CharPtr     # Load address of CharPtr
    sw   zero, 0(t0)       # Set CharPtr to zero

    # Reset the BytePtr to zero
    la   t0, BytePtr      # Load address of BytePtr
    sw   zero, 0(t0)      # Set BytePtr to zero
```

```
        # Reset the WordPtr to zero
        la   t0, WordPtr     # Load address of WordPtr
        sw   zero, 0(t0)      # Set WordPtr to zero

        la t0, TwoDWordPtr
        sw zero, 0(t0)

    Free_Backwards_Loop:
        sb   zero, 0(t1)      # Set the current memory location to zero
        addi t1, t1, -1     # Move the pointer back by one word (4 bytes)
        bge  t1, t2, Free_Backwards_Loop  # Continue loop if still within allocated ra

    Free_End:
        ret       # Return from function
```

- **Memory Calculating Function:**

```
    MemoryCalculated:
        la t0, Sys_MyFreeSpace
        la t1, Sys_TheTopOfFree
        lw t2, 0(t1)
        sub a0, t2, t0
        ret
```

- **String Copy Function:**

```
    strcpy:
        li a5, 1
        la t0, CharPtr
        la t1, CopyPtr
        lw t0, (t0)
    copy_loop:
        lb t4, 0(t1)
        sb t4, 0(t0)

        beq a5, s6 done

        addi a5, a5, 1
    addi t1, t1, 1    # Move to the next byte in B
    addi t0, t0, 1   # Move to the next byte in A

    j copy_loop    # Repeat the loop

done:
    # The copy is complete, and program ends here
    jr ra
```

2 Digital Clock

This task involves creating a digital clock system displayed on 7-segment LEDs. The system will allow users to interact with it to display various time-related information and play sounds.

Features

1. **Interactive Display Modes:** Use specific keys on a key matrix to toggle between the following display modes:
   - **Key 1:** Display the current hour.
   - **Key 2:** Display the current minute.
   - **Key 3:** Display the current second.
   - **Key 4:** Display the current date.
   - **Key 5:** Display the current month.
   - **Key 6:** Display the last two digits of the current year.
2. **Real-Time Updates:** Use a timer to continuously update the displayed time and the 7-segment LED outputs.
3. **Sound Notifications:** Play a sound every minute when the second value reaches 0.
4. **System Calls for Time and Sound:** Research and utilize appropriate system calls to fetch the current time and enable sound playback.

2.1 Project Objectives

The objectives of this project are as follows:

1. Display real-time information, including hours, minutes, seconds, date, month, and year, on 7-segment LEDs.
2. Enable user interaction via a key matrix to toggle between different display modes.
3. Implement a timer to update the displayed information in real-time.
4. Play a notification sound at regular intervals (e.g., when seconds reset to zero).
5. Utilize system calls to fetch current time and generate sound output effectively.

2.2 Methods and Algorithms

The main idea is that when we use the Syscalls a7 30 for time, the output is calculated in milliseconds and give the overflow result when storing in 32 bits register. To deal with it, we store the output in the form of double data type - which has 64 bits available free memory for storing. First, we separate the output by divide it into 2 halfs, the first 32 bits is stored in $a1$ and the remaining 32 bits stored in $a0$. Now, to concate these 2 halfs, we convert in from unsigned integer to double. Next we shift left a1 for 32 bits by multiplying a1 with 2^{32} (This can be done through a for loop, iteration for 32 loops to get $a1 * 2^{32}$). Finally we concate by adding the two results $a1 * 2^{32} + a0$. The keypoint of this methods is that we dealing every single operation in the field of floating point number, which allow us for an easier storing and easy to convert back to integer. In the code, we make use of some function:

1. **main**:
   - Main program loop that handles time calculation and polling the keypad.
   - Calls calculate_time to update time values and check if the second is zero, triggering an alert sound if necessary.
   - Polls the keypad to check for user input.
   - Based on the key pressed, it calls specific methods to display time (hour, minute, second, day, month, year).
2. **calculate_time**:
   - A critical method that performs various calculations to determine the current time, day, month, year, etc.
   - This method involves multiple calculations using both constants (like seconds in a minute, days in a month) and system time values fetched via ecall.
   - This method calculates the current time and the number of seconds since January 1, 1970 (UNIX epoch), and updates the s0, s1, etc., registers with calculated time values.
3. **check_pad**:
   - This method checks for key presses from the user using the keypad. It writes to the input address and then reads the output address to detect whether a key has been pressed.
   - If a key is detected, it jumps to handle_key.
4. **handle_key**:
   - Handles different key presses to perform actions like displaying the hour, minute, second, day, month, or year on the seven-segment display.

- Each key corresponds to a specific display function (e.g., press 1 to display hour, 2 for minute, etc.).

5. **minute_alert**:

   - This method is called when the second value is zero, which could represent the start of a new minute.

6. **encode_display**:

   - This method encodes the time value to be displayed on the seven-segment displays.

2.3 Implementations

2.3.1 Memory-Mapped I/O Address Definition

- Memory Configuration

```
.eqv IN_ADDRESS_HEXA_KEYBOARD 0xFFFF0012
.eqv OUT_ADDRESS_HEXA_KEYBOARD 0xFFFF0014
.eqv SEVENSEG_LEFT  0xFFFF0011
.eqv SEVENSEG_RIGHT 0xFFFF0010
```

- List of digits from 0 to 9 used for seven-segment display.

```
.eqv SEG_0  0x3F
.eqv SEG_1  0x06
.eqv SEG_2  0x5B
.eqv SEG_3  0x4F
.eqv SEG_4  0x66
.eqv SEG_5  0x6D
.eqv SEG_6  0x7D
.eqv SEG_7  0x07
.eqv SEG_8  0x7F
.eqv SEG_9  0x6F
```

2.3.2 Data section

```
SEG_MAP:    .byte SEG_0, SEG_1, SEG_2, SEG_3, SEG_4, SEG_5, SEG_6, SEG_7, SEG_8, SEG_9
namthuong:  .word 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
namnhuan:   .word 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
```

- namthuong: number of days of each month in a normal year
- namnhuan: number of days of each month in a leap year

2.3.3 Main Program flow

1. Initialization
   - The program starts by initializing memory-mapped addresses and jumping into a polling loop to:
     - Update the current time.
     - Check for keypad input.
     - Play sound when seconds reset to zero.

   ```
   li a5, IN_ADDRESS_HEXA_KEYBOARD    # Use s1 instead of s1
   li a6, OUT_ADDRESS_HEXA_KEYBOARD   # Use s2 instead of s2
   ```

2. **Polling Loop** The program continuously checks for:
   (a) **Time Updates**:
   It calls calculate_time to fetch and calculate the current time components.
   (b) **Sound Playback**
   - If the seconds (s10) are zero and sound hasn't been played, it calls play_sound to play a sound using the syscall.
   - The sound_played flag is updated to prevent duplicate sounds.

   (c) **Keypad Input**
   - Rows of the keypad are activated and read to detect any pressed keys.
   - Each key corresponds to a display command (e.g., display hours, minutes, etc.).

```
polling:
    # First, calculate and update all time values
    jal calculate_time
    # Check if seconds is 0 to play sound
    beqz s10, minute_alert
    j check_keypad

minute_alert:
    li a7, 31
    li a0, 69
    li a1, 100
    li a2, 7
    li a3, 50
    ecall
    j polling
```

```
check_keypad:
    # Check row 1
    li t0, 0x01
    sb t0, 0(a5)
    lb t1, 0(a6)
    bne t1, zero, handle_key

    # Check row 2
    li t0, 0x02
    sb t0, 0(a5)
    lb t1, 0(a6)
    bne t1, zero, handle_key

    j polling  # If no key pressed, continue polling
```

2.3.4 Keypad Input Handling

```
handle_key:
    # If press 1 (0x21)
    li t0, 0x21
    beq t1, t0, display_hour
    # If press 2 (0x41)
    li t0, 0x41
    beq t1, t0, display_min
    # If press 3 (0xffffff81)
    li t0, 0xffffff81
    beq t1, t0, display_sec
    # If press 4 (0x12)
    li t0, 0x12
    beq t1, t0, display_day
    # If press 5 (0x22)
    li t0, 0x22
    beq t1, t0, display_month
    # If press 6 (0x42)
    li t0, 0x42
    beq t1, t0, display_year

    j polling
```

The program compares keypad input values to predefined constants and triggers corresponding actions:

Key Value	**Action**
`0x21`	Display Hour
`0x41`	Display Minute
`0xFFFFFF81`	Display Second
`0x12`	Display Day
`0x22`	Display Month
`0x42`	Display Year

Each action jumps to a display function (e.g., `display_hour`).

2.3.5 Time Calculation (`calculate_time`)

The `calculate_time` function retrieves real-time clock data and breaks it into time components. The steps are:

1. **Initialization:**
   - Load constants such as seconds in a minute, hour, day, and average month, and the reference year (1970).
   - Convert constants to double-precision floating-point values for subsequent calculations.
2. **Calculate Total Days:**
   - Divide the total number of seconds by the number of seconds in a day:

   $$\text{Days} = \frac{\text{Total Seconds}}{\text{Seconds per Day}}$$

   - Extract the integer part of the result using conversion from floating-point to integer.
3. **Break into Days, Months, Years:** The program calculates:
   - Number of complete days.
   - Current year, accounting for leap years.
   - Current month and day using the `namthuong` or `namnhuan` table.
4. **Extract Time Components:** Remaining seconds are broken into:
   - Hours (`s8`).
   - Minutes (`s9`).
   - Seconds (`s10`).
   - Years (`s5`).
   - Months (`t3`).
   - Days (`t6`).

2.3.6 Display Functionality

```
    display_hour:
    mv t1, s8
    j encode_display

display_min:
    mv t1, s9
    j encode_display

display_sec:
    mv t1, s10
    j encode_display

display_day:
    mv t1, t6
    j encode_display

display_month:
    addi t1, t3, 1      # Add 1 back since we subtracted it earlier
    j encode_display

display_year:
    li a0, 100
    rem t1, s5, a0
    j encode_display

encode_display:
    # Split number into digits and encode for seven-segment display
    li t2, 10
    div t3, t1, t2     # Get tens digit
    rem t4, t1, t2     # Get ones digit

    # Display tens digit on left display
    la t5, SEG_MAP
    add t5, t5, t3
    lb t6, 0(t5)
    li t0, SEVENSEG_LEFT
    sb t6, 0(t0)

    # Display ones digit on right display
    la t5, SEG_MAP
```

```
add t5, t5, t4
lb t6, 0(t5)
li t0, SEVENSEG_RIGHT
sb t6, 0(t0)

j polling
```

The program uses the `encode_display` function to display a value on two seven-segment displays. Steps are:

1. **Split Value into Digits:** Tens and ones digits are extracted using division and modulo.
2. **Map Digits to Encoding:** The digits are converted to seven-segment encoding using `SEG_MAP`.
3. **Write to Displays:** The encodings are written to `SEVENSEG_LEFT` (tens place) and `SEVENSEG_RIGHT` (ones place).

2.3.7 Sound Playback

```
play_sound:
# Save registers that will be used
addi sp, sp, -4
sw ra, 0(sp)

# Play sound using syscall
li a7, 31
li a0, 69        # Pitch
li a1, 100       # Duration
li a2, 7         # Instrument
li a3, 50        # Volume
ecall

# Restore registers
lw ra, 0(sp)
addi sp, sp, 4
jr ra
```

The `play_sound` function uses a syscall to produce a sound when seconds reset to zero. The parameters are:

- **Pitch:** 69

- **Duration:** 100 ms
- **Instrument:** 7
- **Volume:** 50

This ensures an audible alert for the start of a new minute.

3 Simulation Results

Note: The code of problem 17 will cause delay if running at speed of 30 ins/sec. However, max speed may cause lag and some unknown errors. ***CLICK ME***