Computer Architecture CSF342
Lab sheet 5

Topic: Dynamic Space Allocation

1. Understanding syscall 9 (sbrk) in MIPS

What is syscall 9?

In MIPS assembly, syscall 9 is used for dynamic memory allocation, similar to the `malloc()` function in C. It allocates memory from the heap and returns a pointer to the beginning of the allocated block.

How to use syscall 9:

1. Load the number of bytes to allocate into `$a0`
2. Load the value 9 into `$v0` (indicating the sbrk syscall)
3. Execute the `syscall` instruction
4. The address of the allocated memory will be returned in `$v0`

C malloc vs MIPS syscall 9 comparison:

C code example (allocating a node structure):

```
struct Node {
    int data;
    struct Node* next;
};

struct Node* create_node(int value) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = value;
    new_node->next = NULL;
    return new_node;
}
```

Equivalent MIPS code:

```
create_node:
    # Allocate memory for Node structure (8 bytes: 4 for data, 4 for next pointer)
    li $a0, 8           # Size of Node structure
    li $v0, 9           # syscall 9 (sbrk)
    syscall

    # $v0 now contains the address of allocated memory
    # Initialize the node
    sw $a1, 0($v0)      # Store data value (passed in $a1)
    sw $zero, 4($v0)    # Set next pointer to NULL

    jr $ra              # Return with node address in $v0
```

□2. Memory Alignment in MIPS

What is Memory Alignment?

Memory alignment refers to arranging data in memory at addresses that are multiples of the data's size. MIPS architecture requires that:

- Words (4 bytes) must be aligned to addresses divisible by 4
- Halfwords (2 bytes) must be aligned to addresses divisible by 2

Consequences of Misalignment:

- Runtime errors (address errors)
- Performance degradation
- Incorrect data access

Ensuring Proper Alignment:

1. Always allocate memory in multiples of 4 for word data
2. Use the `.align` directive in data sections
3. When working with structures, ensure proper padding
4. For syscall 9, the returned address is always word-aligned

Example of alignment issue and solution:

```
□# Incorrect - may cause alignment error
la $t0, array
lw $t1, 1($t0)  # Loading word from address not divisible by 4

# Correct - properly aligned
la $t0, array
lw $t1, 0($t0)  # Loading word from address divisible by 4
```

□3. Sample Program: Dynamic Matrix Allocation

The following program asks for an integer 'n', allocates n^2 words of memory, fills it with numbers 1 to n^2, and prints them in a matrix format.

```
□.data
    prompt: .asciiz "Enter the value of n: "
    space: .asciiz " "
    newline: .asciiz "\n"
    error_msg: .asciiz "Invalid input! n must be positive.\n"

.text
    .globl main

main:
    # Print prompt
    li $v0, 4
    la $a0, prompt
    syscall

    # Read integer n
```

```
li $v0, 5
syscall
move $s0, $v0        # $s0 = n

# Validate input
blez $s0, invalid_input

# Calculate n^2
mul $s1, $s0, $s0    # $s1 = n^2

# Calculate memory needed (n^2 * 4 bytes)
sll $a0, $s1, 2      # $a0 = n^2 * 4
li $v0, 9            # syscall 9 (sbrk)
syscall
move $s2, $v0        # $s2 = base address of allocated memory

# Fill the matrix with values 1 to n^2
li $t0, 1            # $t0 = counter (1 to n^2)
move $t1, $s2        # $t1 = current address
fill_loop:
    sw $t0, 0($t1)     # Store current value
    addiu $t0, $t0, 1  # Increment counter
    addiu $t1, $t1, 4  # Move to next word
    ble $t0, $s1, fill_loop

# Print the matrix
move $t0, $s2        # $t0 = current address
li $t1, 0            # $t1 = row counter
li $t2, 0            # $t2 = column counter
print_loop:
    lw $a0, 0($t0)     # Load value to print
    li $v0, 1
    syscall

    # Print space
    li $v0, 4
    la $a0, space
    syscall

    addiu $t0, $t0, 4  # Move to next element
    addiu $t2, $t2, 1   # Increment column counter

    # Check if we need a newline
    bne $t2, $s0, same_line
    li $v0, 4
    la $a0, newline
    syscall
    li $t2, 0          # Reset column counter
    addiu $t1, $t1, 1   # Increment row counter
    same_line:

    # Check if we've printed all elements
    mul $t3, $t1, $s0
    add $t3, $t3, $t2
    blt $t3, $s1, print_loop
```

```
    # Exit program
    li $v0, 10
    syscall

invalid_input:
    li $v0, 4
    la $a0, error_msg
    syscall
    j main
```

4. Student Task 1: Ask the user for a number and then allocate space for that many real numbers in the RAM and in a loop ask the user for the numbers. Then calculate the mean and standard deviation of the numbers.

```
# code snippet to store an int into a float register
.....
intVal .word 25
......
lw   $t0, intVal        # $t0 = 25
mtc1 $t0, $f4           # move $t0 → $f4
cvt.s.w $f4, $f4        # $f4 = float($f4) -- type cast

# Print float value
li   $v0, 2            # syscall: print float
mov.s $f12, $f4        # move float into $f12 (print register)
syscall
```

5. Student Task 2: Linked List in Reverse

C Code:

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

int main() {
    struct Node* head = NULL;
    int i, num;

    // Read 10 numbers and create linked list
    for (i = 0; i < 10; i++) {
```

```
        printf("Enter number %d: ", i+1);
        scanf("%d", &num);

        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = num;
        newNode->next = head;
        head = newNode;
    }

    // Print the list (will be in reverse order)
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");

    return 0;
}
```

□Hint for MIPS Implementation:

- Use syscall 9 to allocate memory (8 bytes per node: 4 for data, 4 for next pointer)
- Maintain a pointer to the head of the list
- For each new number, create a node, set its data, and link it to the previous head

6. Student Task 3: Binary Search Tree (Advanced level, Desirable, not Essential)

C Code:

```
□#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a node in BST
struct Node* insert(struct Node* root, int value) {
    if (root == NULL) {
        return createNode(value);
    }
```

```
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }

    return root;
}

// Inorder traversal
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Preorder traversal
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// Postorder traversal
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

int main() {
    struct Node* root = NULL;
    int n, i, value;

    printf("Enter the number of nodes: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("Enter value %d: ", i+1);
        scanf("%d", &value);
        root = insert(root, value);
    }

    printf("Inorder: ");
    inorder(root);
    printf("\n");

    printf("Preorder: ");
```

```
    preorder(root);
    printf("\n");

    printf("Postorder: ");
    postorder(root);
    printf("\n");

    return 0;
}
```

Sample code for insertion of binary tree and in order traversal:

```
################################################################################
#
# MIPS Assembly Program for Binary Search Tree (BST) - 7th VERSION
#
# This program does the following:
# 1. Asks the user for the number of elements (N).
# 2. Loops N times to accept integers from the user.
# 3. Inserts each integer into a Binary Search Tree.
# 4. Performs a recursive inorder traversal to print the tree's elements,
#    which results in the sorted list of numbers.
#
# Node Structure (12 bytes):
#   +0: integer data (payload)
#   +4: address of the left child node
#   +8: address of the right child node
#
# A NULL pointer is represented by the address 0.
#
################################################################################

.data
    root: .word 0                      # Global variable for the address of the root
node. Initially NULL.

    # --- Strings for Prompts and Output ---
    prompt_count: .asciiz "Enter the number of elements: "
    prompt_node:  .asciiz "Enter an integer: "
    space:        .asciiz " "
    newline:      .asciiz "\n"

.text
.globl main

main:
    # --- Get the number of nodes to insert (N) ---
    li $v0, 4                         # syscall for print_string
    la $a0, prompt_count              # load address of prompt string
    syscall

    li $v0, 5                         # syscall for read_integer
    syscall
    move $s0, $v0                     # store N in a saved register $s0

    # --- Setup for the loop ---
```

```
    ### FIX 1: Use a saved register ($s2) for the loop counter. ###
    # This prevents it from being overwritten by function calls.
    li $s2, 0                         # initialize loop counter i = 0 in $s2
    la $s1, root                      # load the address of the global 'root' pointer
into $s1

# --- Loop N times to get numbers and insert them into the BST ---
loop_insert:
    beq $s2, $s0, end_loop_insert     # if (i == N), exit the loop

    # --- Prompt for and read the next integer ---
    li $v0, 4                         # syscall for print_string
    la $a0, prompt_node               # load address of prompt string
    syscall

    li $v0, 5                         # syscall for read_integer
    syscall
    move $a0, $v0                     # move the read integer into $a0 to pass as an
argument to insertNode

    # --- Call the insert function ---
    jal insertNode

    addi $s2, $s2, 1                  # i++
    j loop_insert

end_loop_insert:
    # --- Perform and print inorder traversal ---
    lw $a0, ($s1)                     # load the address of the root node itself into
$a0
    jal inorder

    # --- Print a final newline for clean output ---
    li $v0, 4
    la $a0, newline
    syscall

    # --- Exit the program ---
    li $v0, 10                        # syscall for exit
    syscall

# =============================================================================
# FUNCTION: insertNode
# DESCRIPTION: Inserts a new node into the BST.
# ARGUMENTS:
#   $a0: The integer value to be inserted.
# =============================================================================
insertNode:
    # --- Save registers that will be modified ---
    subi $sp, $sp, 4
    sw $a0, 0($sp)                    # ### FIX 2: Save the integer value to be inserted
###
                                      # This prevents the syscall from overwriting it.

    # --- Step 1: Allocate memory for the new node (12 bytes) ---
```

```
    li $v0, 9                          # syscall for sbrk (allocate memory)
    li $a0, 12                        # specify 12 bytes
    syscall
    move $t0, $v0                     # $t0 now holds the address of the new node

    lw $a0, 0($sp)                    # ### FIX 2: Restore the integer value from the
stack ###
    addi $sp, $sp, 4

    # --- Step 2: Initialize the new node ---
    sw $a0, 0($t0)                    # new_node->data = value
    sw $zero, 4($t0)                  # new_node->left = NULL (0)
    sw $zero, 8($t0)                  # new_node->right = NULL (0)

    # --- Step 3: Find the correct position and insert ---
    lw $t1, root                      # load the value of the root pointer (address of
the first node)
    bne $t1, $zero, insert_loop_start # if (root != NULL), start the search loop

    # --- If tree is empty, new node becomes the root ---
    sw $t0, root                      # root = address_of_new_node
    jr $ra                            # return

insert_loop_start:
    move $t2, $t1                     # $t2 is our 'current' pointer, starting with the
root address
find_spot:
    lw $t3, 0($t2)                    # $t3 = current_node->data

    # Compare new value ($a0) with current node's data ($t3)
    # if (new_value <= current_node->data), go left
    ble $a0, $t3, go_left

# --- Go Right Path ---
go_right:
    lw $t4, 8($t2)                    # $t4 = current_node->right
    bne $t4, $zero, update_current_right # if (right child exists), move to it
    sw $t0, 8($t2)                    # else, link new node here
    jr $ra                            # insertion complete, return

update_current_right:
    move $t2, $t4                     # current = current->right
    j find_spot                       # continue searching

# --- Go Left Path ---
go_left:
    lw $t4, 4($t2)                    # $t4 = current_node->left
    bne $t4, $zero, update_current_left # if (left child exists), move to it
    sw $t0, 4($t2)                    # else, link new node here
    jr $ra                            # insertion complete, return

update_current_left:
    move $t2, $t4                     # current = current->left
    j find_spot                       # continue searching
```

```
# ==============================================================================
# FUNCTION: inorder
# DESCRIPTION: Recursively traverses the tree and prints nodes in inorder.
# ARGUMENTS:
#   $a0: Address of the current node to process.
# ==============================================================================
inorder:
    # --- Function Prologue: Save context on the stack ---
    subi $sp, $sp, 8                    # make space for 2 words (return address, current
node address)
    sw $ra, 4($sp)                      # save return address ($ra)
    sw $a0, 0($sp)                      # save current node's address ($a0)

    # --- Base Case: If current node is NULL, return ---
    beq $a0, $zero, inorder_return

    # --- 1. Recursive call on the left child ---
    lw $a0, 4($a0)                      # $a0 = current_node->left
    jal inorder

    # --- 2. Visit (Print) the current node's data ---
    lw $a0, 0($sp)                      # restore current node's address from stack
    lw $t0, 0($a0)                      # $t0 = current_node->data

    li $v0, 1                           # syscall to print integer
    move $a0, $t0
    syscall

    li $v0, 4                           # syscall to print string
    la $a0, space
    syscall

    # --- 3. Recursive call on the right child ---
    lw $a0, 0($sp)                      # restore current node's address from stack
    lw $a0, 8($a0)                      # $a0 = current_node->right
    jal inorder

inorder_return:
    # --- Function Epilogue: Restore context from the stack ---
    lw $ra, 4($sp)                      # restore return address
    lw $a0, 0($sp)                      # restore original $a0
    addi $sp, $sp, 8                    # deallocate stack space
    jr $ra                              # return to caller
```

□

Appendix: C to MIPS Compilation Protocols

General Compilation Steps:

1. **Function Prologue**:

- Save return address ($ra) and frame pointer ($fp) on stack
- Adjust stack pointer to allocate space for local variables
- Set frame pointer to current stack position

2. **Function Body**:

    - Map local variables to stack locations relative to $fp
    - Pass arguments in $a0-$a3, additional arguments on stack
    - Use $t registers for temporary values (caller-saved)
    - Use $s registers for values that persist across calls (callee-saved)

3. **Function Epilogue**:

    - Place return value in $v0 (and $v1 if needed)
    - Restore saved registers from stack
    - Restore return address and frame pointer
    - Adjust stack pointer to deallocate stack frame
    - Return using jr $ra

Stack Frame Structure:

```
□Higher addresses
|---------------|
| ...           |
| Argument 5    | ← $fp + 16
| Argument 4    | ← $fp + 12
| Saved $ra     | ← $fp + 8
| Saved $fp     | ← $fp + 4
| Local var 1   | ← $fp
| Local var 2   | ← $fp - 4
| ...           |
|---------------| ← $sp
Lower addresses
```

□Calling Convention:

1. Caller saves any $t registers that need to be preserved
2. Caller places first 4 arguments in $a0-$a3, remaining on stack
3. Caller uses jal to jump to function
4. Callee saves $ra, $fp, and any $s registers it will modify
5. Callee executes function code
6. Callee places result in $v0-$v1
7. Callee restores saved registers and stack pointer
8. Callee returns with jr $ra

Memory Allocation:

- Use syscall 9 to dynamically allocate memory
- Remember to free memory when done (syscall 10 not typically used in student code)
- For linked structures, maintain pointers to allocated memory

Important Tips:

- Always preserve $s registers across function calls
- $t registers can be used without preservation but are not guaranteed to persist across calls
- Keep track of stack pointer adjustments to maintain proper stack alignment
- Use the frame pointer to access arguments and local variables consistently