ECE 30
Introduction to Computer Engineering
Programming Project: Fall 2022
"Quick Sort" using Linked List
Oct. 24, 2022

Project TA: Qihuang Chen
Qihuang may hold more office hours based on his schedule and will make an announcement on canvas if so. You can also post your questions on pizza. You can expect to get responses within two days through the piazza. Please start soon.

1 Project Description

The goal of the project is to write a program that implements a quick sorting algorithm on a linked list in the LEGv8 assembly language. Overall, the program will import an array of data and create a singly linked list based on its values. Then the array will be printed in its unsorted state, after which the array will be sorted and reprinted in **ascending** order. You must use **project_starter_code.s** provided in the course canvas as the template file. **Notice the implementation of our code is a bit different from normal quick sort. Hence, the runtime of our implementation isn't nlogn. The reason is to let you practice writing the assembly of double recursive functions. Your goal is to translate C code in the following pages to LEGv8 assembly.**

A **linked list** is a chain of nodes, much akin to a string of pearls on a necklace. Figure 1 shows an example of a singly-linked list. Each node has at least two elements: first, an element that stores the data value, and, second, an element that stores the address of the next node in the list. The head of a linked list is the first node in said list and the tail of a linked list is the last node (see: figure 2). **Given the tail does not have a next node, it points instead to NULL, which is defined as the number zero.** One can only progress from the head of a singly-linked list to its tail. **This project will use a singly-linked list, although the use of linked lists is not a prerequisite to implement the quick sort algorithm.**

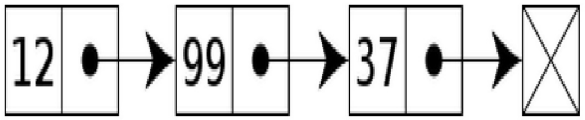

Figure 1: Singly-linked list. Numbers entered by the user are: 12, 99, 37

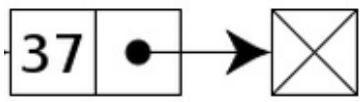

Figure 2: Singly-linked list with a single node. The value at that node is 37 and the node pointer is null, i.e., it does not point to another node. In programming, a null pointer typically is one that is set to 0.

2 Quick Sort Algorithm Description

The overview of a quick sort algorithm is conceptually simple:

1. Get the end node of the input list. We will consider the value of this node the "pivot" value.
2. Partition the input list based on the pivot value so that nodes on the left side have values less than the pivot value and nodes to the right side have value greater or equal to the pivot value. The boundary between left and right is not in the middle. It is computed and retained for next steps.
3. Recursively call quick sort on the left and right sub-lists, respectively. Use the boundary computed in the previous step to distinguish left and right.
4. Return the now-sorted list to the calling function.

The recursive quick sort will keep dividing the sub-list until the sub-list holds a single element. A single element sub-list (the base case) is inherently sorted. It is highly recommended that you read over the description of a quick sort on quick sort on Wikipedia (it's a very good overview), and graphical view of the sort at: www.sorting-algorithms.com/quick-sort.

Your project **must use the provided template file**. While you are free to add any further helper functions, your code must implement the following functions according to specification given below.

Interoperability. The grader should be able to remove any of these functions (and any non-essential helper functions it may use) from your code and use it in another person's programming assignment without issue (i.e. do not change the register names passed to functions).

3 Implementation

In this project, you have to write six functions, namely ***SwapNodeValue***, ***GetLastNode***, ***GetNodeWithVal***, ***Parition***, ***QuickSort***, and ***QuickSortWrapper*** to implement the quick sort algorithm in the LEGv8 assembly language. Some details to note:

- Use the procedures prototype as mentioned below and given to you in the template. **Don't change** the registers or the arguments passed to the procedures or the values returned.
- Follow the "Procedure Call Convention" for calling procedures, passing registers and managing the stack. The procedures should not make any assumptions about the implementation of other procedures, except for the name of input/output registers (as indicated below) and the conventions mentioned in the course. A code that works correctly but does not follow conventions will be penalized.
- We expect your code to be well-commented. Each instruction should be commented with a meaningful description of the operation. For example, this comment is bad as it tells you nothing:

```
// x1 gets x2 - 1
sub x1, x2, #1
```

A good comment should explain the meaning behind the instruction. A better example would be:

```
// Initialize loop counter x1 to n-1
sub x1, x2, #1
```

In this project we use the following c-like structure to implement linked-list nodes:

```
struct Node {
    __int64 data; // the numeric value held in this node
    Node* next; // address of the next node (0 for NULL)
};
```

In the following pesudo-codes, we will also use some c-like notations.

For a node with memory address p:

- p→data: The integer value stored in memory address p
- p→next: The memory address of the next element stored in memory address p+8

In other words, if p is stored in x0, then you can get p→value in x11 and p→next in x12 using the following instructions:

```
LDUR x11, [x0, #0]
LDUR x12, [x0, #8]
```

3.1 Function 1: SwapNodeValue(n1, n2)

The function is to swap data values in two given addresses.

3.1.1 Parameters

- x0: The address of (pointer to) one node (corresponding to `n1`) on the linked list.
- x1: The address of (pointer to) another node (corresponding to `n2`) on the linked list.

3.1.2 Return Value

- Don't need to return anything.

3.1.3 C code

```
void swapNodeValue(struct Node* n1, struct Node* n2) {
    __int64_t temp = n1->data;
    n1->data = n2->data;
    n2->data = temp;
}
```

3.1.4 Example

Before start:

- linked list: 1→4→3→5→2→NULL

Input:

- x0: address of node with value 1
- x1: address of node with value 2

When exiting:

- linked list: 2→4→3→5→1→NULL

3.2 Function 2: GetLastNode(head)

The function uses the given node to find the last node on the linked list and return its address.

3.2.1 Parameters

- x0: The address of (pointer to) a node (corresponding to `head`) on the linked list.

3.2.2 Return Value

- x1: The address of the last node of the linked list.

3.2.3 C code

```
struct Node* getLastNode(struct Node* head) {
    if (head == NULL || head->next == NULL) {
        return head;
    }
    return getLastNode(head->next);
}
```

3.2.4 Example

Before start:

- linked list: 1→4→3→5→2→NULL

Input:

- head: address of a node on the linked list

Output:

- x1 = address of the node (with value 2)

3.3 Function 3: GetNodeWithVal(cur, val)

This is a recursive function to get the address of the node with the given value. **Note: you have to implement it the same recursive way as the C code instead of writing a for loop to get the right node.** The precondition of this function is that the given linked list has a node with the given value.

3.3.1 Parameters

- x0: The address of the node (corresponding to `cur`) of the input list.
- x1: The value (corresponding to `val`) of the node it's looking for.

3.3.2 Return Value

- x2: The address of (pointer to) the node with the given value(x1).

3.3.3 C code

```
struct Node* getNodeWithVal(struct Node* cur, __int64_t val) {
    if (cur != NULL && cur->data != val) {
        return getNodeWithVal(cur->next, val);
    }
    return cur;
}
```

3.3.4 Example

Before start:

- linked list: 1→4→3→5→2→NULL

Input:

- cur: address of a node (with value 1) of the linked list
- val: 5

Output:

- x2: holds the address of the node (with value 5)

3.4 Function 4: Partition(first, lastVal)

This function rearranges the list elements so that all left nodes have values less than the given last node value and all right nodes have values larger than or equal to the given last node value.

3.4.1 Parameters

- x0: The address of the first node (corresponding to `first`) of the linked list.
- x1: The value (corresponding to `lastVal`) of the last node of the linked list.

3.4.2 Return Value

- x2: The address of the first node on the left of the node with the given last node value.

3.4.3 C code

```
struct Node* parition(struct Node* first, __int64_t lastVal)
{
    // Get first node of given linked list
    struct Node* pivot = first;
    struct Node* cur = first;
    struct Node* last = getNodeWithVal(first, lastVal);
    while (cur != NULL && cur != last) {
        if (cur->data < last->data) {
            pivot = first;
            swapNodeValue(first, cur);
            // Visiting the next node
            first = first->next;
        }
        // Visiting the next node
        cur = cur->next;
    }
    swapNodeValue(first, last);
    return pivot;
}
```

3.4.4 Example

Before start:

- Linked list: -3→2→-6→1→NULL

When exiting:

- Linked list: -3→-6→1→2→NULL

Input:

- first: first node(with value -3) address of the list
- lastVal: the value (1) of the last node.

Output:

- x2: the address of the first node(with value -6) on the left of the node(with value 1).

3.5 Function 5: QuickSort(first, last)

This function sorts a linked-list from **first node input** to **last node input**. The given two input nodes are on the same linked list.

3.5.1 Parameters

- x0: The address of the first node (corresponding to `first`) of the list.
- x1: The address of the last node (corresponding to `last`) of the list.

3.5.2 Return Value

- x2: The address of the first node of the list.

3.5.3 C code

```
struct Node* quick_sort(struct Node* first, struct Node* last)
{
    if (first == last) {
        return first;
    }
    struct Node* pivot = parition(first, last->data);
    if (pivot != NULL && pivot->next != NULL) {
        quick_sort(pivot->next, last);
    }
    if (pivot != NULL && first != pivot) {
        quick_sort(first, pivot);
    }
    return first;
}
```

3.5.4 Example

Note that it's a recursive function. In our tests, we may only sort a part of the list instead of the entire list by calling this function. The following example only sorts the list from the first three nodes given node with value 1 and node with value 3. The end result is not entirely sorted.
Before start:

- input list: 1→4→3→5→2→NULL

Input:

- first: first node(with value 1) address of the list
- last: last node(with value 3) address of the list

When exiting:

- partially sorted list: 1→3→4→5→2→NULL

Output:

- x2: first node(with value 1) address of the list

3.6 Function 6: quickSortWrapper(first)

This function sorts a given linked list.

3.6.1 Parameters

- x0: The address of the first node (corresponding to `first`) of the list.

3.6.2 Return Value

- x1: The address of the first node of the list.

3.6.3 C code

```
struct Node* quick_sort_wrapper(struct Node* first) {
    struct Node* last = getLastNode(first);
    return quick_sort(first, last);
}
```

3.6.4 Example

Before start:

- input list: 2→4→3→5→1→NULL

When exiting:

- sorted list: 1→2→3→4→5→NULL

Input:

- first: first node(with value 1) address of the list

Output:

- x1: first node(with value 1) address of the list

4 Instructions

- You must submit your solution on Canvas in a completed file ABC_XYZ_2022_project.s, where ABC and XYZ are the email addresses exluding @ucsd.edu for each student. For example, if Sherlock Holmes and John Watson were working together on a submission, the file name would be: sholmes_jwatson_2022_project.s
- Fill the names and PID of each student in the file.
- Each project team contributes their own unique code – no copying, cheating, or hiring help. We will check the programs against each other using automated tools. These tools are VERY effective, and cheaters WILL get caught!
- Please start this project early! You are provided with a file named **basic_test.data** to test your code. You need to understand the main function and how tests work.

- Try to test the behavior of each function independently, rather than trying to code all of them at once. A good way to test your code is to modify main function and test functions one by one. It will make later debugging far easier. **Also, we will test functions individually. You can assume that all nodes on the same list in our test cases have distinct values.**

5 Grading

- SwapNodeValue (1 point): 1 test case, 1 point each
- GetLastNode (2 points): 2 test cases, 1 point each
- GetNodeWithVal (2 points): 2 test cases, 1 point each
- Parition (2 points): 2 test cases, 1 point each
- Quick_sort (2 points): 2 test cases, 1 point each
- QuickSortWrapper (2 points): 2 test cases, 1 point each
- Assembly flow (5 points): Your code does not compile, run, or terminate correctly. Or, if your code doesn't follow the calling convention or work with our solution code.

Note: Your comments will not be graded, but your TA may refuse to proofread your code if your code is not well-commented.

6 Miscellaneous (please read)

- You MUST show up in YOUR week 5 lab session. Otherwise, your partner has the right to request a new partner. If you cannot make it, notify project TA (Qihuang Chen) in advance.
- Qihuang's office hour: 1-2:30 pm Thursday
- If you get stuck on any function, reviewing the following lecture video might help, especially S4E4-S5E5, S6E6 for recursion (also available on Canvas).

  `youtube.com/watch?v=myYu7DXsQs4&list=PLFxZj9wj5LQwxnB3hpi124YcAYpMDABnf`
- If encountering a syntactic error, Legiss will report the instruction that caused the error in the bottom of the left panel (see also the bottom-right panel). This information is often helpful.
- There is a BUG in LEGv8: You cannot have comments right after the instruction. You must use a SPACE to separate them!

  ```
  LDUR x11, [x0, #0]// Causing a crash
  LDUR x11, [x0, #0] // Okay
  ```
- Complaints about teammates will NOT be considered after the beginning of week 8.