HW2 (Programming Part) Report

> In case VSCode does not render this markdown file correctly, please refer to `report.pdf` for correctly rendered content.

Overview

My assembly code implementation for quicksort follows the pseudocode provided by Wikipedia as shown below:

```
algorithm sort(A, count) is
  quicksort(A, 0, count-1)

algorithm quicksort(A, lo, hi) is
  if lo >= 0 && hi >= 0 && lo < hi then
    p := partition(A, lo, hi)
    quicksort(A, lo, p)
    quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
  pivot := A[lo]
  i := lo - 1 j := hi + 1
  loop forever
    do i := i + 1 while A[i] < pivot
    do j := j - 1 while A[j] > pivot
    if i >= j then return j
    swap A[i] with A[j]
```

The Procedure `swap`

Description

When called by `swap(addr1, addr2)`, it simply exchanges the doublewords stored in addresses `addr1` and `addr2`.

Implementation

Temporary registers x5 and x6 are used to store the 8-byte values in `addr1` and `addr2`

respectively. Then, the values of x5 and x6 are stored to `addr2` and `addr1` respectively, thereby swapping the doublewords in both addresses.

The Procedure `partition`

Description

When called by `partition(addr, lo, hi)`, it first selects `addr[lo]` is the pivot and uses 2 indices, which are `i` and `j` in the pseudocode, to sacn the array `addr[lo:hi]` from left to right and from right to left, respectively. (We refer to these 2 indices as the left index and the right index respectively in the context below.)
The left index (resp. the right index) stops scanning whenever it stops at an element greater (resp. less than) or equal to the pivot. If the indices do not cross (i.e., the left index is still to the left of the right index), swap the elements at the left and right indices and then continue scanning. Otherwise, return the right index `j`.
After the procedure returns, $\mathbf{addr}[m] \leq \mathbf{addr}[n]$ for any pair (m, n) of integers where $m \in [\mathtt{lo}, p], n \in [p+1, \mathtt{hi}]$ and p is the return value of `partition(addr, lo, hi)`.

Implementation

- Line 16~25 extend the stack by 56 bytes to store the values of 6 saved registers (namely x18~x23) and the return address.
- Line 28~30 copy the arguments of the `partition` procedure (which is called by `quicksort`) stored in x10~x12 to saved registers x18~x20, since x10~x12 are used to pass function arguments and will be (mostly) overwritten later in the same procedure while the arguments of `partition` have to be used after `swap` calls.
- Line 33~35 calculate the address of the pivot in memory, store it in temporary register x5, and then load the pivot value, which will be used later in the procedure, to saved register x21.
- Line 38~39 calculate the left the right indices (which are `i` and `j` in the pseudocode respectively) and save the values to saved registers x22 and x23 respectively.

The Outer Loop

The First Inner While Loop

- Line 45 corresponds to `i := i + 1` in the pseudocode.
- Line 47~49 calculate the address of `addr[i]` (i.e., `addr + i * 8`) in memory, store it in temporary register x5, and then load `addr[i]` to the same register. (`i` and `j` mean the left and right indices respectively here and in the context below.)
- Line 50 checks if the condition ($\mathbf{addr[i]} < pivot$) of the first while loop is met. If true, branch to the start of the loop labeled by `innerwhile1`.

The Second Inner While Loop

- Line 54 corresponds to `j := j - 1` in the pseudocode.
- Line 56~58 calculate the address of `addr[j]` (i.e., `addr + j * 8`) in memory, store it in temporary register x5, and then load `addr[j]` to the same register.
- Line 59 checks if the condition ($\texttt{addr[j]} > pivot$) of the second while loop is met. If true, branch to the start of the loop labeled by `innerwhile2`.

The Rest of The Outer Loop

- Line 63~76 correspond to `if i >= j then return j` in the pseudocode.
  - Line 63 checks if `i >= j`. if true, then brach to `exit`, which marks the location of the first instruction corresponding to `swap A[i] with A[j]` in the pseudocode.
  - Line 64 stores the value of x23, which is the right index `j`, to x10, implying that the right index will be returned by this procedure later.
  - Line 68~75 restore the saved registers and the return address that we have backed up in line 19~25. Also, the stack pointer is restored (i.e., the stack frame of the current procedure is popped).
  - Finally, line 76 returns to the caller.
- Line 80~88 correspond to `swap A[i] with A[j]` in pseudocode.
  - Line 80~81 calculate the address of `addr[i]` (i.e., `addr + i * 8`) in memory and store it in register x10, which should store the first argument of the following call to `swap`.
  - Similarly, line 84~85 calculate the address of `addr[j]` (i.e., `addr + j * 8`) in memory and store it in register x10, which should store the second argument of the following call to `swap`.
  - Line 88 calls `swap` using unconditional jump to swap the values of `addr[i]` and `addr[j]`.
- Line 91 jumps back to the start of the outer loop labeled by `outerwhile`.

The Procedure `quicksort`

Description

When called by `quicksort(addr, lo, hi)`, it checks if the values of `lo`, `hi` are valid and whether the subarray `addr[lo:hi]` has more than 1 element. If true, it firstly partition the subarray `addr[lo:hi]` by a call to `partition` and then sort the left and right partitions respectively by further calls to itself. After both partitions are sorted, the subarray becomes sorted as well.

Implementation

- Line 99~106 extend the stack by 40 bytes to store the values of 4 saved registers (namely x18~x21) and the return address.
- Line 109~111 copy the arguments of the `quicksort` procedure (which is called by `sort`) stored in x10~x12 to saved registers x18~x20, since x10~x12 are used to pass function arguments and will be overwritten later in the same procedure while the arguments of the current procedure have to be used later.
- Line 114~ 116 check if the condition of the if statement are met. This corresponds to `if lo >= 0 && hi >= 0 && lo < hi then` in the pseudocode. If there exists an unsatisfied condition, then the program branches to the `exit` label, which marks the head of the instructions responsible of returning to the caller.
- Line 119~120 calls the procedure `partition` and then copy the return value (denoted by `p` in the later context) stored in x10 to saved register x21 as `p` will be used later. This corresponds to `p := partition(A, lo, hi)` in the pseudocode. Note that we do not have to alter the values (arguments) of x10~x12 before the call to `partition` given that the call has exactly the same arguments as the current `quicksort` procedure and that the values of x10~x12 remain unmodified since the start of the current procedure.
- Line 123~126 store arguments to x10~x12 and then call the procedure `quicksort` as a subroutine. This corresponds to `quicksort(A, lo, p)` in the pseudocode.
- Line 129~132 store arguments to x10~x12 and then call the procedure `quicksort` as a subroutine. This corresponds to `quicksort(A, p + 1, hi)` in the pseudocode.
- Line 136~141 restore the saved registers and the return address that we have backed up in line 102~106. Also, the stack pointer is restored (i.e., the stack frame of the current procedure is popped).
- Finally, line 142 returns to the caller.

The Procedure `sort`

Description

When called by `sort(addr, count)`, it sort the array `addr[0:count-1]` by calling `quicksort(addr, 0, count-1)`.

Implementation

- Line 148~149 extend the stack by 8 bytes to store the return address.
- Line 152~154 calculate `count-1` (which is stored in x12), store arguments to x11~x12, and then call the procedure `quicksort`. This corresponds to `quicksort(A, 0, count-1)` in the pseudocode. Note that we do not have to alter the value (argument) stored in x10 before the call to `quicksort` given that first argument of the call is exactly the same as that of the current `sort` procedure and that the value of x10 remains unmodified since the start of the current procedure.
- Line 157~158 restore the return address that we have backed up in line 149. Also, the

stack pointer is restored (i.e., the stack frame of the current procedure is popped).
- Finally, line 159 returns to the caller, `main` procedure.