**The MlPS stack: recap**





* The stack grows downward in terms of memory addresses.
* The address of the top element of the stack is stored (by convention) in the "stack pointer" register, Ssp.

O($sp), 1($sp), ... are "used" locations

-1($sp), -2($sp), ... are "free"

* MIPS does not provide "push" and "pop" instructions. Instead, they must be done explicitly by the programmer.
* "push" $t0 simulated with:

sub $sp, $sp, 4 # $t0 needs 4 bytes sw $t0, 0($sp) # write to stack

* "pop" $t0 simulated with:

lw $t0, 0($sp) # read stack top addi $sp, $sp, 4 # free 4 bytes

Ox?FFFFFFF

$sp

stack

OxOOOOOOOO .....\_ ,

## Using the stack: MP 2's main



* Performs a jal,so must save $ra before and restore it afterwards

main:

addi $sp, $sp, -4 # grow stack

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SW** | $ra, 0($sp) | # | save | callee-saved register | $ra |
| jal | iterTraverse | # | call | your function |  |
| lw | $ra, 0($sp) | # | restore $ra | | |
| addi  Jr | $sp, $sp, 4  $ra | # | shrink stack | | |

A j a1 to a function

Will cause a malfunction If ra is not on the stack

And caller-saved regs And lastly, no messes

Should be handled like eggs In each of the esses

They're junk when the function comes back Make sure you restore 'em - don't slack!

# Practice with pointers: Linked Lists



* + Linked lists are implemented in C using structures as follows: struct node {

int data; node"' next;

}

// data field

// pointer to next node ,n list

* + If p is a *pointer* to a node, then

' P is the node itself

(' P).data is located at address p in memory

(' p). next is located at address p + si zeof (data) in memory

* + If p points to the last node in the list, (\*p) . next == NULL.
* If to points to a node, then the statement to = ("'tO). next makes to point to the next node in the list.
  + The above statement can be translated into MIPS as:

lw $t0, 4($t0)

* + Translate the following C statements into MIPS: (\*tO).data = (\*tl).data;

(\*tO).next = (\*tl).next;

## MIPS functions to traverse lists



### Translate this into MIPS:

void printList(node\* p) { while(p != NULL) { //

print((\*p).data); p = c 'p).next;

}

}

# BAD code! printList:

beq $a0, $0, PL\_done

# need to save $ra, $a0 lw $a0, 0($a0)

jal print

# need to restore $a0 lw $a0, 4($a0)

j printList

# need to restore $ra PL\_done:

Jr $ra

**NULL== 0**

# GOOD code printList:

beq $a0, $0, PLdone addi $sp, $sp, -8

sw $ra, 0($sp) sw $a0, 4($sp) lw $a0, 0($a0) jal print

lw $ra, 0($sp) lw $a0, 4($sp) addi $sp, $sp, 8 lw $a0, 4($a0) J printList

PL\_done: jr $ra

**A more efficient solution**



printList:

addi $sp, $sp, -8 sw $ra, 0($sp)

sw $s0, 4($sp) move $s0, $a0

PL\_loop:

beq $s0, $0, PL\_done lw $a0, 0($s0)

jal print

lw $s0, 4($s0)

j **PL\_loop**

PL\_done:

# loop body has

# only the 5

# "necessary"

# instructions

Jr $ra

|  |  |  |
| --- | --- | --- |
| lw | $ra, | 0($sp) |
| lw | $s0, | 4($sp) |
| addi  . | $sp, | $sp, 8 |

# Recursive list traversal



### Translate this into MIPS:

void printList(node\* p) { if(p != **NULL)** {

print((\*p).data); printList((\*p).next);

}

}

# BAD code printList:

beq $a0, $0, PL\_done

lw $a0, 0($a0) jal print

lw $a0, 4($a0) jal printList

PL\_done: jr $ra

# correct, but messy printList:

beq $a0, $0, PLdone addi $sp, $sp, -8

sw $ra, 0($sp) sw $a0, 4($sp) lw $a0, 0($a0) jal print

lw $a0, 4($sp) lw $a0, 4($a0) jal printList lw $ra, 0($sp) addi $sp, $sp, 8

PL\_done: jr $ra

## A simple way to do recursion



recursive(args) { if(base\_condition) {

*II* base case stuff

return;

}

else { *II* !base\_condition

*II* recursive case

}

}

recursive:

b\_not\_base\_condition recursive\_case

# base case stuff jr $ra

recursive\_case:

# grow stack to save $ra (at least)

jal recursive

# shrink stack to restore $ra + other stuff Jr $ra

# Recursive list traversal



**Translate this into MIPS:**

int printList(node\* p) { if(p == NULL)

return 0; else {

print((\*p).data);

return 1 + printList((\*p).next);

# Recursive solution printList:

bne $a0, $0, recurs,ve\_case move $v0, $0

jr $ra

recurs1ve\_case: addi $sp, $sp, -8 sw $ra, 0($sp) sw $a0, 4($sp)

}

} lw $a0, 0($a0)

jal print

lw $a0, 4($sp) lw $a0, 4($a0) jal pri ntList addi $v0, $v0, 1

lw $ra, 0($sp) addi $sp, $sp, 8

Jr $ra

# Recursive list reverse-traversal



### Translate this into MIPS:

printListReverse:

void printListReverse(node\* p) { if(p == **NULL)**

printNewline(); else {

printListReverse((\*p).next); print((\*p).data);

bne addi

SW

jal lw

a.ddi Jr

$a0, $0, PL\_recursive

$sp, $sp, -4

$ra, 0($sp) printNewline

$ra, 0($sp)

$sp, $sp, 4

$ra

} PL\_recursive:

} addi $sp, $sp, -8

sw $ra, 0($sp) sw $a0, 4($sp) lw $a0, 4($a0)

jal printListRecursive lw $a0, 4($sp)

lw $a0, 0($a0) jal print

lw $ra, 0($sp) addi $sp, $sp, 8 Jr $ra