

Function Calls

Tom Kelliher, CS 240

Feb. 16, 2004

1 Administrivia

Announcements

Homework due Wednesday.

Assignment

Read 3.7–9.

From Last Time

SPIM lab.

Outline

1. Function calls: stack execution model, memory use, call/return convention, example.

Coming Up

Operand addressing, program life cycles.

2 Function Calls

Consider the following code fragment:

```
int min(int, int);
int factorial(int);

void main(void)
{
    int i = 3;
    int j = 5;
    int k;

    k = min(i + 3, j);

    /* Current values of i, j, k? */

    k = factorial(5);
}

int min(int a, int b)
{
    int i;

    if (a < b)
        i = a;
    else
        i = b;

    b = 0;

    return i;
}

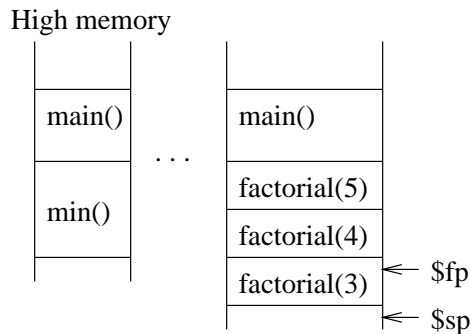
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Notes:

1. Caller/callee.
2. Call by value parameters. Formal/actual. Other call mechanisms: address, reference, name, etc.
3. How are the values passed? How is the return value returned? How is memory allocated?
4. How are the recursive invocations of `factorial()` managed?

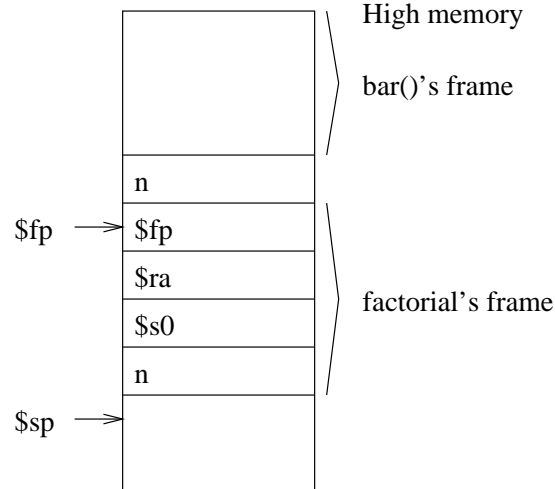
2.1 Stack Execution Model

1. Maintain a stack of active function invocations:



- (a) Stack frame.
 - (b) Contents of a frame: local variables, saved registers, return address, frame pointer of caller, actual parameters.
Scratch-pad storage on top of the stack.
 - (c) Frame pointer and stack pointer.
 - (d) Caller vs. callee save.
2. More detail: Suppose `main` calls `factorial`, using an on-stack parameter (`n`) pass.

`factorial` uses `$s0` and has a local copy of `n` (inefficient).



Example code snippets:

```
lw $s0, 4($fp)
```

```
sw $s0, -12($fp)
```

```
sw $s0, 0($sp)
```

```
sub $sp, $sp, 4
```

3. Register usage convention:

- (a) `$v[0-1]` — function results.
- (b) `$a[0-3]` — function arguments.
- (c) `$t[0-9]` — caller save registers. (Except for two global pointers, ignore these registers.)
- (d) `$s[0-7]` — callee save registers.
- (e) `$sp` — top of stack pointer, pointing to first free location on execution stack.
- (f) `$fp` — frame pointer, pointing to base of current frame.
- (g) `$ra` — return address.

2.2 Memory Use

1. Upon entry to `main`, the frame stack is ready to go.
Program parameters.
2. `main` pushes its frame and gets to work.
3. Upon exit, `main` restores `$ra` (among others) and executes `jr $ra`.

2.3 Call/Return Conventions

Use this sequence. Assume caller is calling callee.

Caller:

1. Store any caller save registers.
2. Store any register parameters.
3. Push any on-stack parameters (caller and callee must agree on order).
(C: last pushed first.)
4. Execute `jal callee`

Callee:

1. Push stack frame by subtracting size of frame from `$sp`.
Number of words needed:
 - (a) One each for `$fp` and `$ra`.
 - (b) One for each `$sx` register used.
 - (c) One for each parameter and each local variable.

2. Save registers (`$fp`, `$ra`, callee save), using `$sp` as base register.
3. Set `$fp` by adding frame size to `$sp` and storing.
4. (Optional) Copy parameters into frame.
5. Body of callee executes. Frame references should use `$fp` as base register.
6. Store return value in appropriate `$ax` registers.
7. Restore saved registers, using `$sp` as base register.
8. Pop frame from stack by adding frame size to `$sp`.
9. Execute `jr $ra`

Caller:

1. Collect and store return values.

2.4 Example Code

1. Consider the following C++ code which recursively computes the factorial:

```
#include <stdio.h>

int getnum(void);
int factorial(int n);

int main()
{
    int value;

    printf("Enter 0 to exit.\n");

    value = getnum();

    while (value != 0)
    {
```

```

        printf("Factorial: %d\n", factorial(value));
        value = getnum();
    }

    return 0;
}

int getnum(void)
{
    int num;

    printf("Number: ");
    scanf("%d", &num);
    return num;
}

int factorial(int n)
{
    if (n <= 1)
        return 1;

    return n * factorial(n - 1);
}

```

2. Write MIPS program which passes parameters in registers.

3. Frame maps:

main	getnum	factorial
\$fp	\$fp	\$fp
\$ra	\$ra	\$ra
\$s0	num	\$s0
value		n

4. Here is the corresponding SPIM program:

```

# factorial.spim --- A recursive SPIM program. Demonstrates function
# call and return.

```

```

        .data
prompt:  .ascii "Number: "
nl:      .ascii "\n"
instr:   .ascii "Enter 0 to exit.\n"
response: .ascii "Factorial: "

#####
# main
#####

        .text
        .globl main
main:
        sub $sp, $sp, 16      # Push frame & save registers.
        sw $fp, 16($sp)
        sw $ra, 12($sp)
        sw $s0, 8($sp)
        add $fp, $sp, 16

        li $v0, 4            # Print instruction.
        la $a0, instr
        syscall

        jal getnum           # Get a number from keyboard.

        sw $v0, -12($fp)     # Store locally.
        move $s0, $v0

while1:
        beqz $s0, endwhile1 # Compute until 0 entered.

        li $v0, 4            # Print response prompt.
        la $a0, response
        syscall

        move $a0, $s0        # Pass argument

        jal factorial        # Call

        move $a0, $v0        # Copy return value to print it
        li $v0, 1
        syscall

        li $v0, 4            # Prepare to read next number.

```

```

    la $a0, nl
    syscall

    jal getnum

    sw $v0, -12($fp)    # Store number just read.
    move $s0, $v0

    b while1

endwhile1:
    lw $s0, 8($sp)     # Restore registers and pop frame.
    lw $ra, 12($sp)
    lw $fp, 16($sp)
    add $sp, $sp, 16
    li $v0, 0
    jr $ra             # Exit.

#####
# getnum --- returns integer read from keyboard through $v0.
#####

    .text
getnum:
    sub $sp, $sp, 12    # push frame & save registers.
    sw $fp, 12($sp)
    sw $ra, 8($sp)
    add $fp, $sp, 12

    li $v0, 4
    la $a0, prompt
    syscall

    li $v0, 5
    syscall             # Value read left in $v0.

    sw $v0, -8($fp)

    lw $ra, 8($sp)     # restore registers & pop frame.
    lw $fp, 12($sp)
    add $sp, $sp, 12
    jr $ra

```

```
#####
# factorial --- compute factorial of $a0 and return result through
#   $v0
#####
```

```

        .text
factorial:
        sub $sp, $sp, 16
        sw $fp, 16($sp)
        sw $ra, 12($sp)
        sw $s0, 8($sp)
        add $fp, $sp, 16

        sw $a0, -12($fp)      # Our value n.
        move $s0, $a0

        bgt $s0, 1, recurse  # Base case. Return 1
        li $v0, 1
        lw $s0, 8($sp)
        lw $ra, 12($sp)
        lw $fp, 16($sp)
        add $sp, $sp, 16
        jr $ra

recurse: sub $a0, $a0, 1      # Recursive call. Compute (n-1)!
        jal factorial
        mul $v0, $v0, $s0    # n * (n-1)!
        lw $s0, 8($sp)
        lw $ra, 12($sp)
        lw $fp, 16($sp)
        add $sp, $sp, 16
        jr $ra

```