Function Calls & Stack Frames

- Topics to be covered:
  - Function calls
  - How function calls are implemented
  - Stack frames
  - Example: factorial.c

- Notes:
  - MIPS assembly language is used to illustrate the detailed implementation of stack operations by a C compiler such as gcc
  - MIPS is used as a representative language for describing the detailed operations
  - In COMP1921 you are expected to develop an understanding that C is translated into assembly code before it can be executed and relate the assembly code implementation to the detailed operations performed by a processor; you will not be required to write assembly code

Slide credits: Alan Blair
Functions

What exactly happens when we execute...

L1:
    \[ z = f(x, y); \]
L2:
    \[ \ldots \]

1. values of \( x \) and \( y \) are passed to \( f \)
2. control is transferred to function \( f \)
3. local variables of \( f \) are created
4. code for \( f \) executes
5. return value is set up
6. local variables of \( f \) are removed
7. return value is copied into \( z \)
8. execution resumes at statement \( L_2 \)

In this lecture, we examine how each of these steps are performed in detail
Function Calls

- First consider how to transfer control to a function and then back again.
- When we go to $f$, we need to remember where to return to.
- The “jump and link” instruction `jal` provides this:
  
  ```
  L1: ...
  jal f # Reg[31] ← &L2 (store return address)
  # PC ← &f (fetch next instruction from f)
  L2: ...
  ```

- How to get back:
  
  ```
  jr $31 # PC ← Reg[31] (fetch next instruction from return address)
  ```
A potential problem: \texttt{f} calls \texttt{g}.

\begin{verbatim}
L1: ...
   jal f  # main calls "f"
L2: ...

f: ...
   jal g  # "f" calls "g"
L3: ...
   jr $31 # what value in $31?

g: ...
   ...    # code for "g"
   jr $31
\end{verbatim}

So \texttt{f} needs to save its “return address”.

In another register?
That’s ok ... unless \texttt{f} calls \texttt{g} calls \texttt{h} calls ...
Function Calls

If main calls $f$ calls $g$ calls $h$ …
…then $h$ finishes, then $g$ finishes, then $f$ finishes and we’re back in main

Function call/return uses last-in, first-out (LIFO) protocol ⇒ use a stack of return addresses
Implementing Function Calls

Where/how to implement this stack?
At the top of memory, using the $sp register.

...  
li $sp, 0xfffffffffc  
...

f:  
sub $sp, $sp, 4  # save return-  
sw $ra, 4($sp)  # address on stack  
...  # ... code for f ...
lw $ra, 4($sp)  # restore return-  
add $sp, $sp, 4  # address from stack  
jr $ra  # return to caller

and similarly for g and h.

“Indexed addressing mode”: Add index (4) to contents of $sp register and use as address
MIPS Memory Map
- where are programs & data stored?

Function parameters, return addresses, and locally declared variables

Data dynamically allocated by your program \((\text{malloc}, \text{etc.})\)

Your program’s global variables are stored here

Words of program code (main and functions) – where PC points to
MIPS Registers

- Simple MIPS processors have 32 x 32-bit registers
  - Conventions for using them have evolved

<table>
<thead>
<tr>
<th>Names</th>
<th>#'s</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>at</td>
<td>1</td>
<td>Reserved for assembler</td>
</tr>
<tr>
<td>v0,v1</td>
<td>2,3</td>
<td>Expression evaluation and results of a function</td>
</tr>
<tr>
<td>a0...a3</td>
<td>4..7</td>
<td>Arguments 1-4</td>
</tr>
<tr>
<td>t0..t7</td>
<td>8..16</td>
<td>Temporary (not preserved across function calls)</td>
</tr>
<tr>
<td>s0..s7</td>
<td>16..23</td>
<td>Saved temporary (preserved across function calls)</td>
</tr>
<tr>
<td>t8,t9</td>
<td>24,25</td>
<td>Temporary (not preserved across function calls)</td>
</tr>
<tr>
<td>k0,k1</td>
<td>26,27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>gp</td>
<td>28</td>
<td>Pointer to global area</td>
</tr>
<tr>
<td>sp</td>
<td>29</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>fp</td>
<td>30</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>ra</td>
<td>31</td>
<td>Return address (used by function calls)</td>
</tr>
</tbody>
</table>

Registers can be referred to by their symbolic name ($ra) or their number ($31)
Functions

What exactly happens when we execute...

L1:
\[
z = f(x, y);
\]
L2:
\[
\ldots
\]

1. values of \( x \) and \( y \) are passed to \( f \)
2. control is transferred to function \( f \)
3. local variables of \( f \) are created
4. code for \( f \) executes
5. return value is set up
6. local variables of \( f \) are removed
7. return value is copied into \( z \)
8. execution resumes at statement L2
Returning a result

Return values are placed in register $v0 by the callee, from where they can be retrieved by the caller. E.g.

```c
int one() {
    return 1;
}

one:
    sub $sp, $sp, 4  # save return-
    sw $ra, 4($sp)   # address on stack
    #
    li $v0, 1  # v0 = 1;
    #
    lw $ra, 4($sp) # restore return-
    add $sp, $sp, 4 # address from stack
    jr $ra          # return v0;
```
Functions

What exactly happens when we execute...

L1:
    \( z = f(x, y); \)
L2:
    ...

1. values of \( x \) and \( y \) are passed to \( f \)
2. control is transferred to function \( f \)
3. local variables of \( f \) are created
4. code for \( f \) executes
5. return value is set up
6. local variables of \( f \) are removed
7. return value is copied into \( z \)
8. execution resumes at statement L2
Passing arguments

- All C function arguments are expressions. To pass them to functions:
  - evaluate each argument
  - put it where the function can find it
- By convention, use registers \$a0..\$a3.
- Extra or large arguments are placed on stack.
- E.g. the function call: \texttt{calc}(5, x+1, 7-y)

```assembly
li \$a0, 5  # first arg
lw \$a1, x
add \$a1, \$a1, 1  # second arg
li \$a2, 7
lw \$t0, y
sub \$a2, \$a2, \$t0  # third arg
jal calc  # function call
```
Preserving registers

- The **caller** needs to ensure that values in registers $a0..a3, t0..t9$ are not lost across calls, if in use:
  - For each of $a0..a3, t0..t9$ that is in use:
    - save it on stack before setting up arguments
    - restore when call returns

- The called function (**callee**) needs to ensure registers $s0..s7$ are preserved across function calls.
  - For each $s0..s7$ that is modified:
    - save it on stack at start of function
    - restore it before returning
Functions

What exactly happens when we execute...

L1:
   \[ z = f(x, y); \]
L2:
   ...

1. values of \( x \) and \( y \) are passed to \( f \)
2. control is transferred to function \( f \)
3. local variables of \( f \) are created
4. code for \( f \) executes
5. return value is set up
6. local variables of \( f \) are removed
7. return value is copied into \( z \)
8. execution resumes at statement L2
Local Variables

- Try to allocate local variables to $s\_? registers.
- If we have:
  - > 8 local variables
  - a local variable > 4-bytes (e.g. array)
we put them in memory ... on the stack.
- Space for these is allocated when the function starts. E.g.

```c
int fun(int x, int y) {
  char a[10], b[20];
  int i,j,k,l,m,n,p,q,r,s,t;
  ... // assembly code writer or compiler puts
    // a,b,r,s,t on stack and rest in $s0-7
}
```
Stack Frames

- Inside a function, we need access to:
  - arguments
  - local variables

- When the function terminates, need to retrieve
  - register values from the caller
  - previous stack pointer
  - return address

- All of these are located on the stack. Thus, a small region on the stack is associated with the invocation of each function.

- This region is referred to as a stack frame
**gcc Stack Frame**

- **Memory**
  - **High address**
  - **Stack grows down**
  - **Set up call in calling function**
  - **Push excess arguments onto stack**
  - **FP value of the caller**
  - **In called function, push return address**
  - **In called function, push any other callee saved registers**
  - **In called function, create space for local variables**

- **Local data for active function**
  - **Stack pointer $sp**
  - **Frame pointer $fp**

- **Points to first word stored by called function**

- **Saved values of registers from calling function**

- **Arguments**

- **Previous frame pointer**

- **Return address**

- **Other registers (such as $s0-7)**

- **Local variables**

- **Stack grows down**

Creating a Stack Frame

On entry to a function:

1. allocate memory for frame by subtracting frame size from $sp
2. save callee-saved registers in frame (at least $fp, $ra, and any $s? registers used by function)
3. set new frame pointer to previous $sp value

E.g. a function $f$ that uses $s0$ only:

```
f:  subu $sp, $sp, 12 # space for $fp, $ra, $s0
    sw $fp, 12($sp)  # save $fp value
    sw $ra, 8($sp)   # save $ra value
    sw $s0, 4($sp)   # save $s0 value
    add $fp, $sp, 12 # set frame pointer
```
Removing a Stack Frame

On exit from a function:
1. ensure that return value is in $v0
2. restore all callee-saved registers
3. pop stack frame by adding frame size to $sp
4. return by jumping to location in $ra

E.g. exit code for f:

```assembly
...               # code for f function
lw $s0, 4($sp)    # restore $s0 value
lw $ra, 8($sp)    # restore $ra value
lw $fp, 12($sp)   # restore $fp value
addu $sp, $sp, 12 # pop stack frame
jr $ra            # return to caller
```
Example: `factorial.c`

```c
int factorial( int n ) {
    printf("n at %X is equal to %d\n", &n, n );
    if (n < 1) {
        return 1;
    } else { 
        return ( n * factorial( n - 1 ));
    }
}

int main (void) {
    int fact; int n;
    printf("Enter number: ");
    scanf( "%d", &n );
    fact = factorial( n );
    printf("Factorial of %d is %d\n", n, fact);
    return 0;
}
```
Enter number: 4
n at BFD8E1B0 is equal to 4
n at BFD8E190 is equal to 3
n at BFD8E170 is equal to 2
n at BFD8E150 is equal to 1
n at BFD8E130 is equal to 0
Factorial of 4 is 24

- In contrast to the MIPS calling convention of passing arguments via registers, the Intel x86 convention is to pass these on the stack. Thus we see decreasing addresses for the argument (n) in the recursive calls to factorial.
Stack Overflow

Occurs when stack space is exceeded due to amount of data (stack frames) stored on the stack

Enter number: 1000000
n at BFD8E1B0 is equal to 1000000
n at BFD8E190 is equal to 999999
n at BFD8E170 is equal to 999998
n at BFD8E150 is equal to 999997

...

n at BF46E700 is equal to 738098
n at BF46E6E0 is equal to 738097
n at BF46E6C0 is equal to 738096
Segmentation fault
Where to from here?

- Next Week
  - Files & File Operations (Moffat, Ch. 11)
  - Number Representation & Bit Operations