Function Call and Return

Anatomy of a Function Call

In a language that features recursion, it is necessary to distinguish between the definition of a function and a call to that function. The definition of a function defines how that function behaves. A call to the function creates an "instance" or "activation" of that function. Although a function has only one definition, over time it will have many distinct instances. For a recursive function, several instances may exist simultaneously.

Each instance has memory requirements. Between the time a function is called and the time it exits, a function's activation record requires memory to contain:

- local storage — parameters, local variables, and any auxiliary space the compiler has determined is necessary to implement that function's block of code.
- return information — where and how execution should continue when this call exits.

The memory block allocated for each activation is called an **activation record**. The memory within an activation record is only valid while the function is executing; once the function exits, the memory is reclaimed.

Activation Records

An activation record stores the state associated with one instance, or call, of a function. Foremost this requires the activation record to reserve space for the parameters and local variables. Depending on the implementation, the activation record will also store ancillary information associated with the call.

The form of the activation record for a function can be deduced at compile time from its declaration. Like a C structure, an activation record is a collection of values each of which has a name and a type. The values are all stored together in a block. The following syntax declares a routine that takes two integers as value parameters:

```
void Simple(int a, int b)  // two value params
{
   int temp1, temp2;       // temporary local variables
   ....
```

From the above declaration, the compiler deduces that the activation record will need space for four integers: **`a`**, **`b`**, **`temp1`** and **`temp2`**. Assuming an integer takes **`4`** bytes, the activation record will take at least **`16`** bytes. The compiler does not allocate an activation

record now, but later when the function is actually called, the compiler will know to allocate that much space for the new function instance.

The Stack

A stack is the perfect structure to allocate and de-allocate activations. When a function is activated, space for one of its activation records is pushed onto the stack and initialized. The activation record must remain intact so long as the function executes. When the function exits, the memory for its activation record can be reclaimed. The activation record is popped from the stack and the previous function's activation will become the one on top of the stack. Since function calls are very common, it's important that the stack manage allocation and de-allocation very efficiently.

```
void A(void)              void B(void)              void C(void)
{                         {                         {
  B();                      C();                      printf("Hi!");
```

There is one constraint about function calls that the stack exploits: functions must exit in the reverse order that they were called. So if function **A** calls **B**, and then **B** calls **C**, then eventually the functions must exit in the order **C**, **B**, **A**. It is not possible for the **B** instance to spontaneously disappear out of turn from between the **A** and **C** instances. Only the instance which is currently executing needs to access its variables; all the other instances are suspended. In this case when **C** prints out **"Hi!"**, **A** and **B** are suspended. **B** is waiting for **C** to exit, and **A** is waiting for **B**.

The "stack" data type is perfectly suited for storing activation records— it gives ready access to the current activation while keeping all the others suspended. The stack is a fairly common abstract data type to maintain a collection like a stack of plates. Only the "topmost" plate is visible. The stack is defined by two operations— Push and Pop. Push takes a new plate and puts it on the top of the stack. The new plate blocks access to all the plates below. Pop removes the top plate, and the plate below moves up to become the new top of the stack.

In the case of function calls, the plates are actually activation records. The topmost plate is the activation record of currently executing function. When a function is called, its activation record is pushed on the stack and the new instance becomes the currently executing function. When the instance exits, the previously executing instance resumes.

Activation Protocol

The code generated by the compiler must implement some consistent protocol for passing control and information from calling function to called function and back. Different languages and even different compilers within a language may use slightly different protocols. Most protocols depend on the caller and callee having access to a "prototype" description of the interface of the routine.

Here is a simple protocol to manage the transfer of parameters and control for a single C language function:

1) The caller pushes space onto the stack for the **parameters** to the call

2) The caller initializes the parameters in the new activation record (copies the values)

3) The caller saves state and return information for execution after call exits. For us, this come as an additional 4 bytes pushed on top of the parameters.

4) The caller suspends, callee pushes space onto stack for local variables and begins executing at the beginning of the function

5) Callee function executes normally, accessing variables, parameters from its activation record

6) When the callee exits, its portion of the activation record is popped from the stack

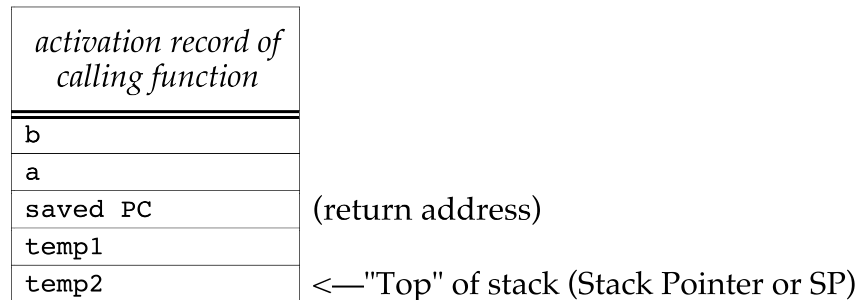
7) Control transfers to saved return address in caller function, who removes the parameters from the stack

Note that in this protocol, local variables are not initialized— the stack pointer is merely extended enough to allocate space for their existence. Parameters look like local variables where the caller has filled in an initial value.

Most computers have highly specialized instructions for operating on stacks since the stack is such a central structure in so many computer activities. As a result, the above protocol can most likely be implemented very efficiently. Some optimizing compilers have schemes where some parameters are passed in registers instead of through the stack, but we will ignore that option for now. To make operations fast, a couple of registers are devoted to various execution housekeeping needs. The **`SP`** register is dedicated to holding the address of the top of the stack. The **`PC`** register holds the address of the currently executing instruction, and the **`RV`** register will be used to communicate the return value from a function. The rest of our registers (**`R1`**, **`R2`**, etc.) will be general-purpose registers, used for all sort of various purposes.

A protocol also needs to specify the general arrangement of space in an activation record. We are going to choose a very simple arrangement for the ease of our discussion. In our protocol, parameters will be pushed on the stack in **`right-to-left`** order, (there is an important reason why it must be right to left for any C compiler which we will learn later), followed by the saved return address, and then the local variables in **`top-to-bottom`** order. We'll assume that the return value from the function won't be placed on the stack, but instead will be written in the special register **`RV`**. Note this won't allow for return values that are larger than the word size of the machine, but usually those are handled in a complicated (and often somewhat

expensive) way which we won't be interested in. As is true on most current architectures, the activation stack will grow downward, from higher addresses down to lower. Given the **Simple** function declaration from page 1, here is its activation record layout:



Given the arrangement of parameters and local variables in our protocol, we can think of an activation record as similar to 2 neighboring **struct**s, one for the local variable block and another block for the parameter, with the saved return address in the middle. The **SP** is pointing to the base address of the local variable block, the **return** address follows, and the parameter block begins right after that.

```
struct SimpleActivationLocal {
   int temp2;  // offset 0 (from SP)
   int temp1;  // offset 4
};

struct SimpleActivationParameters {
   int a;      // offset 12 (from SP)
   int b;      // offset 16
};
```

In compiled code, each function refers to its local variables and parameters by their respective offsets from the end of the stack. For example, in the **Simple** function, the parameter **b** is an offset of **16** bytes from the stack pointer. Notice that:

- The protocol allows the compiler to generate code without knowing exactly where the activation will actually end up in memory at runtime. All variables are identified via their address, which is always the value in the **SP**, plus the relevant offset.

- The code works equally well for any activation of that function.

- No trace of the actual identifiers used in the source is left in the code. The type information for any particular variable is not explicit in the code, but is implicit in the operations the compiler generates to work on that data.

Call and Return

To support function call operations, we add two additional instructions to our instruction set. These specialized instructions direct control flow from one function to another and handle saving and restoring the execution state.

The **`Call`** instruction is similar to a **`Jmp`** in that it immediately redirects control to another location, but it also makes space on the stack and stores the return address, the address of the next instruction to execute when control returns back to the calling function. We indicate the function to jump to by listing its symbolic name. That's actually the way the compiler does it. The linker is responsible for translating that name into the address of the code for that function. The address is what we need at runtime, since at that point we don't know how to find a function by name in memory.

```
Jumps to strlen & saves the next instruction's address on stack
Call  <strlen>
```

The **`Ret`** instruction is usually the last one in the code generated for a function body. After doing the function work, cleaning up its use of the stack, and perhaps storing the return value in the **`RV`** register, the function uses the **`Ret`** instruction to restore execution back to the calling context. When **`Ret`** is called, the **`SP`** is assuming to be pointing to the area on the stack where the return address was saved, so it basically has the effect of changing the **`PC`** to hold that address and then continuing from there. It also pops the space holding the stored PC value off the stack, so that the PC points exactly where it was prior to function call, almost as if the function were never really called at all.

```
Returns control to saved instruction address stored on stack
Ret
```

Formal vs. Actual Parameters

Functions and procedures in structured languages often accept "parameters"— values supplied by the calling code that communicate information to the function body. There is a distinction between these parameters as they exist within the body of the function and the values that are passed to the function when it is called. Parameters in the function declaration and used in the body of a function are known as its "formal parameters". Values passed to a function when the function is called are known as "actual parameters". For example:

```
void Jane(void)
{
   int arf, meow;

   John(arf, meow);
}

int John(int left, int right)
{
   return (left * right);
}
```

In the above example, **left** and **right** are the formal parameters for function **John**. **arf** and **meow** are the actual parameters for **John** as it is called from **Jane**.

Value Parameters

In C, all parameters are passed by **value**. For value parameters, the activation record contains a slot to hold a copy of the formal parameter. The caller just copies the value of the actual parameter into the slot when the function is called. In this way, subsequent changes to the formal parameter are not reflected in the actual parameter.

It may be the case that we want the actuals of a function call to change when the function changes one of its formals. For example:

```
int main(void)
{
   int i,j;
   i = 5;
   j = 76;
   Betty(i,j);
   printf("%d %d\n", i, j);
}

void Betty(int m, int n)
{
   m = 20;
   n = 30;
}
```

Call-by-value produces the output: **5 76**

It may be the case that we want **i** and **j** to change in the body of **Betty**. In C, one accomplishes this using a well-known workaround of passing pointers and modifying pointed-to values. For example, if in C we wanted to allow **Betty** to (indirectly) modify her formals, we can write:

```
int main(void)
{
   int i,j
   i = 5;
   j = 76;
   Betty(&i, &j);
   printf("%d %d\n",  i,  j);
}

void Betty(int *m, int *n)
{
   *m = 20;
   *n = 30;
}
```

Which should yield the output: **20 30**

Source: http://see.stanford.edu/materials/icsppcs107/15-Function-Call-And-Return.pdf