CA4003 - Compiler Construction

Runtime Environments

David Sinclair

Runtime Environments

When procedure A calls procedure B, we name procedure A the *caller* and procedure B the *callee*.

A *Runtime Environment*, also called an *Activation Record*, is a convention that defined the environment in which a procedure is executed. It can be considered as a contract between the *caller* and *callee*. It covers how data is passed from the *caller* to the *callee* and how data is returned from the *callee* to the *caller*. It also describes how local data is stored and managed in a procedure.

Runtime Environments allow:

- separate compilation of procedures;
  - build large programs
  - reasonable compilation times
- the use of libraries.
  - use system libraries
  - combine procedures written in different languages.

Runtime Environments (2)

Unfortunately there is no standard *Runtime Environment*. Each compiler may have its own *Runtime Environment*, but then you can only link with procedures compiled with the same modules. An Operating System vendor can define a *Runtime Environment* for its system libraries. Then a compiler vendor either has to use the same *Runtime Environment*, or write libraries that "convert" between the different *Runtime Environments*.

The key challenges in designing a *Runtime Environment* are:

- to handle nested procedures; and
- be as efficient as possible.

Runtime Environments are very architecture specific. We will focus on an implementation of a MIPS *Runtime Environment* to highlight the major aspects of a *Runtime Environment*.

MIPS Registers

Number	Name	Purpose
0	$0	Always contains the constant 0.
1	$at	*Assembler Temporary*. Used by assembler to expand pseudo-ops.
2-3	$v0-$v1	*Returned Value* of a procedure. If value is contained in one word then only $v0 is significant.
4-7	$0-$a3	*Argument Registers.* These contain the first 4 arguments of a procedure call.
8-15, 24, 25	$t0-$t9	*Temporary Registers*
16-23	$s0-$s7	*Saved Registers*
26-27	$k0-$k1	*Kernel Registers*. **Do not use!**
28	$gp	*Global Pointer*. Used to access global static variables.
29	$sp	*Stack Pointer*
30	$fp	*Frame Pointer*
31	$ra	*Return Address*

Shaded rows must be preserved across procedure calls.

Stack Frames

In most modern programming languages, a procedure can define *local variables* that are created when the procedure in entered and destroyed when the procedure is exited. If a procedure is nested, particularly if it is recursive, there can be several instances of variable with the same name. Each instance is local to the instance of the procedure that declares it.

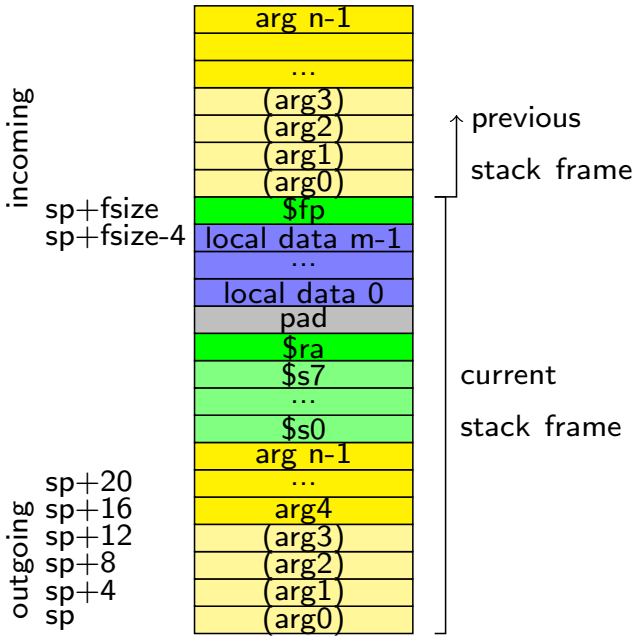
As these instances of the procedures' local variables go out of scope (are destroyed) in *last-in/first-out* manner, a stack is the natural data structure to use to manage the *Runtime Environments*.

Caveat: In languages that support nested functions **and** functions returned as results (higher-order functions), then a local variable may need to exist after its declaring function has returned. In these situations we can't use stacks.

Stack Frames (2)

Each time a procedure is called a *stack frame* is created by the *caller*. The *stack frame* contains:

- Space to store the arguments passed to the procedure.
- Space to save the *saved registers* ($s0-$s7).
- Space to store the procedures *return address*.
- Space for local data.



Stack Frames (3)

Arguments
The stack frame contains space to store the arguments passed to the procedure by the current procedure. The first 4 words (arg0, arg1, arg2 and arg3) are not used by the *caller* as these arguments are passed in registers $a0 to $a3. Space is reserved for them in the stack frame in case the *callee* needs to *spill* them.

While early (70's and 80's) machine passed the arguments on the stack, modern machines use registers as empirical studies have shown that most procedures have 4 or less arguments. But what is the procedure f calls another procedure g? It will have to save its arguments (f's arguments) into the stack frame. Doesn't this lose the time we saved by putting the arguments in registers? Yes, except for *leaf procedures* (procedures that do not call other procedures) and most procedures are leaf procedures.

Stack Frames (4)

Callee Saved Registers
This is space for the *callee* to save the values of any of the saved registers ($s0 to $s7) whose value may change during the execution of the procedure. On exit from the procedure, the *callee* restores the original values of $s0 to $s7.

Return Address
This is used by the *callee* to store the value of the return address register, $ra. It is copied back to $ra just before the *callee* returns.

Pad and Local Data
This is used for local variables and any temporary registers that need to be preserved across procedure calls. The *Pad* is inserted before the *Local Data* to ensure that the *Local Data* starts on a double word boundary (i.e. that the stack frame size is a multiple of 8). Also the *Local Data* needs to be padded to ensure its size is also a multiple of 8.

Frame Pointer

If function g calls function f, then on entry to f, the stack pointer, $sp, points to the first argument that g passed to f. f's first task is to allocate a stack frame by simply subtracting the *framesize* from the stack pointer. The existing *frame pointer*, $fp, is stored in the stack frame and the new $fp then becomes the old stack pointer, $sp, and this is used to access the incoming arguments. When f exists, $fp is copied to $sp and $fp is restored to its previous value.

When the *framesize* is fixed, then we do not need to use a *frame pointer* since $fp = $sp + framesize and we can calculate the constant offsets to the $sp. However, sometimes the *framesize* is not known until late in the compilation process when the number of saved registers and temporaries can be determined. But we may need offset to the formal arguments and local variables earlier much earlier, so we put them near the *frame pointer* at offset that we know.

Static Links

In some *block structured* languages (such as C, java, ML, etc.) inner procedures can use variables of outer procedures. In order to enable a procedure to access the variables of its enclosing procedure, whenever a procedure is called it is passed a pointer to the stack frame of the most recent invocation of the function that **lexically** encloses it. This is called a *static link*. In order to find an outer variable, it may be necessary to "chain" through a series of *static links*.

Dynamic links refer to the most recent invocation of the procedure that invoked the current procedure.

Static Links (2)

Setting static links links:

- If the callee is nested directly within the caller, set its static link to point to the caller's frame pointer (or stack pointer).
- If the callee has the same nesting level as the caller, set its static link to point to wherever the caller's static link points.

There are other ways, such as *displays* and *lambda lifting* to handle block structured languages.

Example

Consider the follow code

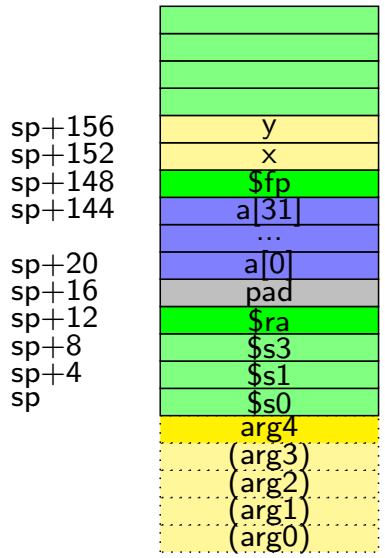
```
int g(int x, int y)
{
  int a[32];

  ... (calculate using x, y, and a)

  a[1] = f(y, x, a[2], a[3], a[4]);
  a[0] = f(x, y, a[1], a[2], a[3]);

  return a[0];
}
```

If only $s0, $s1 and $s3 need to be preserved in *g* it's stack frame is:



Example (2)

The generated MIPS code could be:

```
g:      # start of prologue
        sw      $fp, -4($sp)
        lw      $fp, $sp
        addiu   $sp, $sp, (-152)    # push stack frame
                                    # w/o outgoing arguments

        sw      $ra, 12($sp)        # save return address

        sw      $s0, 0($sp)        # save callee-save registers
        sw      $s1, 4{$sp)
        sw      $s3, 8($sp)
        # end of prologue

        # start of the body of procedure g
        ...

        # save $a0 and $a1 in caller stack frame
        sw      $a0, 0($fp)         # save $a0 (variable x)
        sw      $a1, 4($fp)         # save $a1 (variable y)
```


Example (3)

```
        # first call to procedure f
        lw      $t0, 36($sp)      # arg4 is variable a[4]
        sw      $t0, -4($sp)      # passed in stack frame
        la      $sp, -4($sp)
        lw      $a3, 36($sp)      # arg3 is variable a[3]
        la      $sp, -4($sp)
        lw      $a2, 36($sp)      # arg2 is variable a[2]
        la      $sp, -4($sp)
        lw      $a1, 0($fp)       # arg1 is variable x
        la      $sp, -4($sp)
        lw      $a0, 4($fp)       # arg0 is variable y
        la      $sp, -4($sp)

        jal     f
        la      $sp, 20($sp)      # adjust sp by 4n

        sw      $v0, 24($sp)      # store result of f into a[1]
```

Example (4)

```
# second call to procedure f
lw      $t0, 32($sp)      # arg4 is variable a[3]
sw      $t0, -4($sp)      # passed in stack frame
la      $sp, -4($sp)
lw      $a3, 32($sp)      # arg3 is variable a[2]
la      $sp, -4($sp)
lw      $a2, 32($sp)      # arg2 is variable a[1]
la      $sp, -4($sp)
lw      $a1, 4($fp)       # arg1 is variable y
la      $sp, -4($sp)
lw      $a0, 0($fp)       # arg0 is variable x
la      $sp, -4($sp)

jal     f
la      $sp, 20($sp)      # adjust sp by 4n

sw      $v0, 20($sp)      # store result of f into a[0]
```


Example (5)

```
# load results of g
lw      $v0, 20($sp)       # result is a[0]

# end of the body of procedure g

# start of epilogue
lw      $s0, 0($sp)       # restore callee-save registers
lw      $s1, 4($sp)
lw      $s3, 8($sp)

lw      $ra, 12($sp)       # restore return address

addiu   $sp, $sp, 152      #pop stack frame
lw      $fp, -4($sp)       # retore frame pointer

jr      $ra
```