CS 302 : Implementation of Programming Languages

TUTORIAL 8 (Runtime Environments); April 05, 2017

The generic forms discussed in the notes for calling convention and activation record structure, are given below for reference and are to be used for solving these problems. Activation record (AR) contains the information required for a single activation of a procedure.

1. Local variables 2. Parameters 3. Return address
4. Saved registers 5. Static link 6. Dynamic link 7. Return value

Tasks performed by Caller (Caller prologue protocol)	**Tasks performed by Callee (Callee prologue protocol)**
1. Make space on the stack for a return value. 2. Pass actual parameters through stack or registers. 3. Sets the static link. 4. Jumps to the called function. Return address saved on stack.	1. Sets the dynamic link. 2. Sets base of new activation record. 3. Saves registers on the stack. 4. Makes space for locals on stack.
Tasks performed by Caller (Caller epilogue protocol)	**Tasks performed by Callee (Callee epilogue protocol)**
1. Picks return value from stack	1. Restores registers. 2. Sets base pointer to AR of calling function. 3. Restores Stack pointer to location containing returned value. 4. Returns back to caller

P1. Two functions in C are given to you as shown below. The function main() calls the function g() which in turn calls printf(). The assembly codes are also given alongwith the source code . For the function g(), several lines of assembly code are annotated with the task they perform. The assembly instructions that relate to runtime environment are shown in colour and also indicate the type of protocol and the task number (refer to the generic design of caller-callee conventions outlined above).

(a) Examine the annotations against the assembly code for g() and report errors, if any. Do the necessary corrections if required, and explain why the instructions achieve their intended task. Justify why certain parts of the protocol are missing from the assembly code. Explain the presence of 16 in the following assembly instruction :

subq $16, %rsp # callee prologue - 4

(b) Add to the source code of g() so that the number 16 becomes a much larger number, say 124, or becomes smaller (say 4, if possible).

(c) Add annotations with justifications to the assembly instructions of g() that are left blank.

(d) Insert annotations to the assembly code of main() on the same lines as for g().

ALTERNATELY do the entire exercise of this problem using the compiler on your system and report.

You may note that the leave instruction is a combination of two assembly instructions :

```
movq %rbp, %rsp      # restores stack pointer of caller
popq %rbp            # restores base pointer of caller
```

```
void g( int x, const int y)
{



 x = x * 5;


  if ( x > y ) x++;


  else


     x = x--;


    printf( " x = %d y = %d \n", x, y);


}









main()
{


  int a = 10,


  b = 2;


  g(a+a*b, b);


  a++;


}
```

```
.LC0:
        .string " x = %d y = %d \n"
        .type   g, @function
g:
.LFB0:
        pushq   %rbp                # callee prologue – 1
        movq    %rsp, %rbp          # callee prologue – 2
        subq    $16, %rsp           # callee prologue – 4
        movl    %edi, 4(%rbp)  # load x
        movl    %esi, 8(%rbp)   # load y
        movl     4(%rbp), %edx  # .......
        movl    %edx, %eax          # .......
        sall    $2, %eax            # .......
        addl    %edx, % eax         # .......
        movl    %eax,  4(%rbp)   # x <..
        movl     4(%rbp), %eax   # load x
        cmpl    8(%rbp), %eax   # compare x with y
        jle     .L2                 # if x <= y goto .L2
        addl    $1, 4(%rbp)         # x < x+1
        jmp     .L3                 # jump past else part
.L2:
        subl    $1, 4(%rbp)         # x : else part
.L3:
        movl    $.LC0, %eax         # load format sting
        movl    8(%rbp), %edx   # caller prologue  2
        movl    4(%rbp), %ecx   # load x
        movl    %ecx, %esi          # caller prologue  2
        movq    %rax, %rdi          # caller prologue  2
        movl    $0, %eax            # ........
        call    printf              # caller prologue  4
        leave                       # callee epilogue  2,3
        ret                         # callee epilogue  4
main:
        pushq   %rbp            #
        movq    %rsp, %rbp      #
        subq    $16, %rsp       #
        movl    $10, 4(%rbp)   #
        movl    $2, 8(%rbp)    #
        movl    8(%rbp), %eax  #
        addl    $1, %eax        #
        imull   4(%rbp), %eax  #
        movl    8(%rbp), %edx  #
        movl    %edx, %esi      #
        movl    %eax, %edi      #
        call    g               #
        addl    $1, 4(%rbp)    #
        leave                   #
        ret                    #
```

P2. Consider the C program given below.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct rec { int data;
          struct rec* next;
}rec;

int aa = 5, b = 10;

int sum (int i, int j)
{ static int val = 0;
  char c;
  val = val + i;
  printf( " static area of sum : & val = %p val = %d \n",
&val, val);
   printf (" params of sum : &i = %p; &j = %p \n", &i,
&j);
  printf (" locals of sum : &c = %p \n", &c);
  return i + j;
}

double foo (double);

int main () {
    static int val = 10;
    val++;
    double (*fp1)(double) = &foo;
    int (*fp2)(int, int) =  sum;
    int (*fp3)() = main;

        printf(" globals :   &aa = %p \n &b = %p.
\n",&aa,&b);
     printf( " static area of main : & val = %p val = %d
\n", &val, val);
      printf(" code area1 : fp1 = &foo = %p \t &foo =
%p\n",fp1, &foo);
     printf(" code area 2 : fp2 = &sum = %p \t \n", fp2);

     printf(" code area 3 : fp3 = &main = %p \t \n", fp3);


    int a[10];
    int u = 5, v = 10;
    rec x;
    rec *y;
    y = (struct rec*) malloc (sizeof(struct rec)) ;
    x.next = y;
    printf(" locals of main : &fp1 = %p & fp2 = %p &fp3
= %p\n", &fp1, &fp2, &fp3);
    printf(" locals of main : &a[] %p &a[9] = %p u = %p
& v = %p &x = %p \n", a, &a[9], &u, &v, &x);
     printf(" &x.data = %p &x.next = %p & y = %p \n",
&x.data, &x.next, &y);
    printf(" heap area : y = %p \n", y);
    fp1(100);
    fp2(10, 15);
    return 0;
}

double foo (double x){
    static float val = 0.0;
    char c;
    val += x;
     printf( " static area of foo: & val = %p val = %f \n",
&val, val);
    printf (" params of foo : &x = %p \n", &x);
    printf (" locals of foo : &c = %p \n", &c);
    return x*x;
}
```

(a) Compile and run the program and record the output. Also generate the assembly code for the source.
(b) Correlate the locations of all areas, code and data, used in the program above, as generated by the compiler with the physical addresses in the output.
(c) Identify all the assembly instructions that relate to run-time management in all the functions and annotate them by correlating with the caller-callee conventions and the source code.
(d) Annotate the rest of the assembly code by correlating it with the source code. Also justify why the physical addresses in the output are consistent with the layout of the AR for each function.

P3. Write a program fragment comprising two functions - caller f() and a callee g() that meets the following specifications. The functions g() returns a value of a built-in type and has two formal parameters – a single dimensional array and a scalar of a buit-in type. The function f() calls the function g() from within an expression. Perform the following activities and report your findings.

(a) Compile and generate the assembly code for the source.
(b) Identify all the assembly instructions that relate to run-time management in all the functions and annotate them by correlating with the caller-callee conventions and the source code.
(c) Identify the parameter passing mechanism used and justify the source and corresponding assembly.
(d) Annotate the rest of the assembly code by correlating it with the source code. In particular show how return values are communicated between two functions.
(e) From the analysis and understanding of assembly code, abstract the semantic analysis and intermediate code generation issues for such a call and express the same in English. Write a SDTS so that from the intermediate code produced by you, it should be possible, in principle, to generate assembly code as produced by your compiler.

P4. Do the tasks as asked in P3 above, except for the change in specifications of f() and g() as indicated in the following. Function g() returns a literal of some type, has 2 formal parameters which are a 2D array and a reference to a scalar. The function f() calls g() with appropriate actual arguments from within an expression.

P5. Do the tasks as asked in P3 above, except for the change in specifications of f() and g() as indicated in the following. Function g() returns a struct object, has 2 formal parameters which are an union object and and an array of struct. The function f() calls g() with appropriate actual arguments from within an expression.

P6. Do the tasks as asked in P3 above, except for the change in specifications of f() and g() as indicated in the following. Function g() returns a pointer to a struct object, has 3 formal parameters which are a const object, a pointer to a function and a reference to an struct object. The function f() calls g() with appropriate actual arguments from within an expression.

P7. Do the tasks as asked in P3 above, except for the change in specifications of f() and g() as indicated in the following. Function g() returns a reference to a class object, has 2 formal parameters which are a const class object, and an array of pointers to objects of a certain class. The function f() calls g() with appropriate actual arguments from within an expression. Return value is a class object, 2 parameters are a const class object, reference to a class object.

P8. Do the tasks as asked in P3 above, except for the change in specifications of f() and g() as indicated in the following. There are N (>=2) number of overloaded definitions of the callee g() and M (M >=2) calls to g() with appropriate actual arguments in the body of f(). You are permitted to choose the return types and formal parameters of the overloaded versions of function g(). The focus is on working our the semantic analysis issues related to resolving calls to an overloaded function along with the associated issues of runtime environment.

P9. Write a SDTS for nested procedures that would perform all the runtime environment activities including the setting up of the static links. Define suitable abstractions in your intermediate code for this purpose, if necessary. Augment the SDTS for such procedures discussed in the course so that the generated intermediate code has the prologue and epilogue before a function call and a return after semantic analysis.

Supratim Biswas, Posted on April 5, 2017