Implementing Subprograms

In Text: Chapter 10

Outline

• General semantics of calls and returns
• Implementing "simple" subroutines
• Call Stack
• Implementing subroutines with stack-dynamic local variables
• Nested programs

General Semantics of Calls and
Returns

• The subroutine call and return operations are together called subroutine linkage
• The implementation of subroutines must be based on the semantics of the subroutine linkage

Semantics of a subroutine call

• Save the execution status of the current program unit
• Pass the parameters
• Pass the return address to the callee
• Transfer control to the callee

Storage of Information

• The call and return actions require storage for the following:
  – Status information about the caller
  – Parameters
  – Return address
  – Return value for functions
  – Local variables

Semantics of a subroutine return

• If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters
• Move the return value to a place accessible to the caller
• The execution status of the caller is restored
• Control is transferred back to the caller

General Semantics of Calls and Returns

• The subroutine call and return operations are together called subroutine linkage
• The implementation of subroutines must be based on the semantics of the subroutine linkage
Implementing "simple" subroutines

- **Simple subroutines** are those that cannot be nested and all local variables are static.
- A simple subroutine consists of two parts: code and data:
  - Code: constant (instruction space)
  - Data: can change when the subroutine is executed (data space)
- Both parts have fixed sizes.

Activation Record

- The format, or layout, of the data part is called an **activation record**, because the data is relevant to an activation, or execution, of the subroutine.
- The form of an activation record is static.
- An **activation record instance** is a concrete example of an activation record, corresponding to one execution.

An activation record for simple subroutine

- Since the activation record instance of a "simple" subprogram has fixed size, it can be statically allocated.
- Actually, it could be attached to the code part of the subprogram.

The code and activation records of a program with simple subroutines

- Four program units—MAIN, A, B, and C
- MAIN calls A, B, and C
- Originally, all four programs may be compiled at different times individually.
- When each program is compiled, its machine code, along with a list of references to external subprograms are written to a file.

How is the code linked?

- A linker is called for MAIN to create an executable program.
  - Linker is part of the OS
  - Linker is also called loader, linker/loader, or link editor
  - It finds and loads all referenced subroutines, including code and activation records, into memory.
  - It sets the target addresses of calls to those subroutines' entry addresses.

Assumptions so far...

- All local variables are statically allocated.
- No function recursion.
- No value returned from any function.
Call Stack

- **Call stack** is a stack data structure that stores information about the active subroutines of a program.
- Also known as *execution stack*, *control stack*, *runtime-stack*, or *machine stack*.
- Large array which typically grows downwards in memory towards lower addresses, shrinks upwards.

```
Push(r1):
    stack_pointer--;
    M[stack_pointer] = r1;

r1 = Pop();
    r1 = M[stack_pointer];
    stack_pointer++;
```

Call Stack

- When a function is invoked, its activation record is created dynamically and pushed onto the stack.
- When a function returns, its activation record is popped from the stack.
- The activation record on stack is also called stack frame.
- **Stack pointer** (sp): points to the frame top.
- **Frame pointer** (fp): points to the frame base.

Implementing subroutines with stack-dynamic local variables

- One important advantage of stack-dynamic local variables is support for recursion.
- The implementation requires more complex activation records:
  - The compiler must generate code to cause the implicit allocation and deallocation of local variables.

Dynamic Link (control link) = previous sp

- Used in the destruction of the current activation record instance when the procedure completes its execution.
- To restore the sp in previous frame (caller).
- The collection of dynamic links in the stack at a given time is called the dynamic chain, or call chain, which represents the dynamic history of how execution got to its current position.

More complex activation records

- Since the return address, dynamic link, and parameters are placed in the activation record instance by the caller, these entries must appear first.
- Local variables are allocated and possibly initialized in the callee, so they appear last.
Why do we need dynamic links?

- The dynamic link is required in some cases, because there are other allocations from the stack by a subroutine beyond its activation record, such as temporaries.
- Even though the activation record size is known, we cannot simply subtract the size from the stack pointer to remove the activation record.
- Access nonlocal variables in dynamic scoped languages.

An Example without Recursion

void fun1(float r) {
  int s, t;
  fun2(s);
  ...
}

void fun2(int x) {
  int y;
  ...
  fun3(y);
  ...
}

void fun3(int q) {
  ...
}

void main() {
  float p;
  fun1(p);
}

• Call sequence:
  main → fun1 → fun2 → fun3

• What is the stack content at points labeled as 1, 2, and 3?

Local Variable Allocation

- Local scalar variables are bound to storage within an activation record instance.
- Local variables that are structures are sometimes allocated elsewhere, and only leave their descriptors and a pointer to the storage as part of the activation record.

An Example

void sub(float total, int part) {
  int list[5];
  float sum;
  ...
}

• Function recursion means that a function can eventually call itself.
• Recursion adds the possibility of multiple simultaneous activations of a subroutine at a given time, with at least one call from outside the subroutine, and one or more recursive calls.
• Each activation requires its own activation record instance.
An Example

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

void main() {
    int value;
    value = factorial(3);
}
```

Implementing nested subroutines

- Some static-scoped languages use stack-dynamic local variables and allow subroutines to be nested
  - FORTRAN 95, Ada, Python, and JavaScript
- Challenge
  - How to access nonlocal variables?

Two-step access process

- Find the activation record instance on the stack where the variable was allocated
  - more challenging and more difficult
- Use the `local_offset` of the variable to access it
  - `local_offset` describes the offset from the beginning/bottom of an activation record

Key Observations

- In a given subroutine, only variables that are declared in static ancestor scopes are visible and can be accessed
- Activation record instances of all static ancestors are always on the stack when variables in them are referenced by a nested subroutine: A subroutine is callable only when all its static ancestors are active

Finding Activation Record Instance

- Static chaining
  - A new pointer, static link (static scope pointer or access link), is used to point to the bottom of an activation record instance of the static parent
  - The pointer is used for access to nonlocal variables
  - Typically, the static link appears below parameters in an activation record

Finding Activation Record Instance

- A static chain is a chain of static links that connect the activation record instances of all static ancestors for an executing subroutine
- This chain can be used to implement nonlocal variable access
Finding Activation Record Instance

• With static links, finding the correct activation record instance is simple
  – Search the static chain until a static ancestor is found to contain the variable
• However, the implementation can be even simpler
  – Compiler identifies both nonlocal references, and the length of static chain to follow to reach the correct record

Finding Activation Record Instance

• \texttt{static\_depth} is an integer associated with a static scope that indicates how deeply it is nested in the outermost scope
• The difference between the \texttt{static\_depth} of a nonlocal reference and the \texttt{static\_depth} of the variable definition is called \texttt{nesting\_depth}, or \texttt{chain\_depth}, of the reference
• Each reference is represented with an ordered integer pair (chain\_offset, local\_offset)

An Ada Example

\begin{verbatim}
procedure Main_2 is
  procedure Main_2 is
    procedure Sub3 is
      a : integer := 0;
      begin
        a := 1;
        a := 2;
        a := 3;
      end;
      a := 4;
    end;
  end;
begin
  a := 5;
  a := 6;
  a := 7;
  a := 8;
  a := 9;
end;
end;
\end{verbatim}

Stack Contents

The stack contents for the Ada example are shown in the diagram. The stack contains local variables and procedure calls. The stack frame for each procedure includes the procedure name and the parameters passed to it. The stack also shows the return addresses and the possible stack growth during procedure execution.

What is the static depth for each procedure? What is the representation of A at points 1, 2, and 3?