Implementing Subroutines

In Text: Chapter 9

Outline [1]

- 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

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

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

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

Local variables
Parameters
Return address

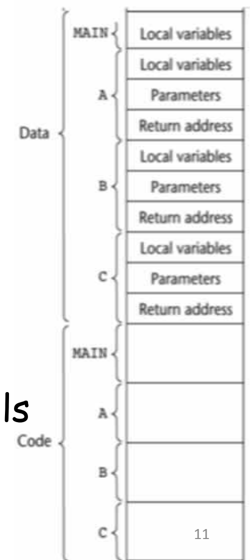
- Since the activation record instance of a "simple" subprogram has fixed size, it can be statically allocated
- Actually, it could be attached to the code part of the subprogram

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



N. Meng, S. Arthur 11

Assumptions so far…

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

N. Meng, S. Arthur 12

Call Stack [2]

- **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

Call Stack

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

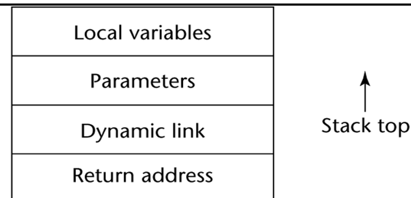
Call Stack

- 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

More complex activation records

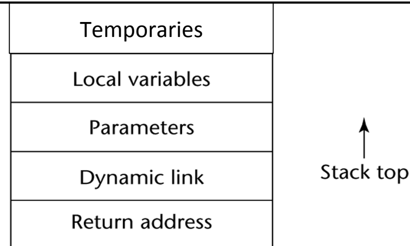


- Since the return address, dynamic link, and parameters are placed in the activation record instance by the caller, these entries must appear first
- Local variables are allocated and possibly initialized in the callee, so they appear last

Dynamic 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

Why do we need dynamic links?

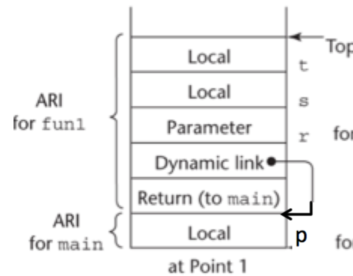


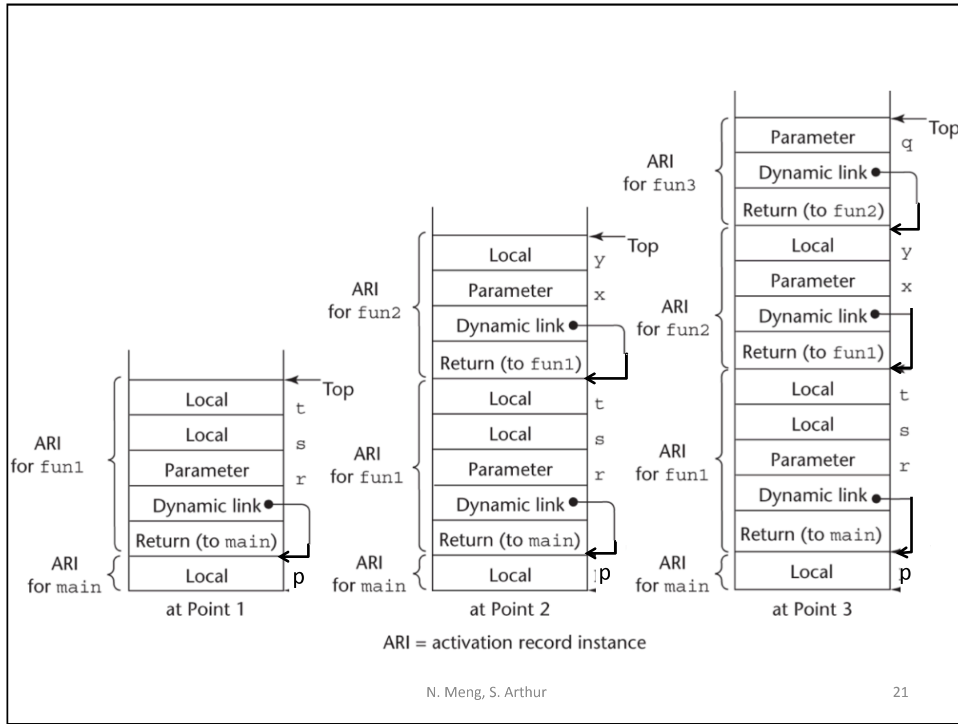
- The dynamic link is required in some cases, because there are other allocations from the stack by a subroutine beyond its activation record, such as temporaries
- Even though the activation record size is known, we cannot simply subtract the size from the stack pointer to remove the activation record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

An Example without Recursion

```
void fun1(float r) {
    int s, t;
    ... <---------1
    fun2(s);
}
void fun2(int x) {
    int y;
    ...  <--------2
    fun3(y);
    ...
}
void fun3(int q) {
    ... <---------3
}
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?




ARI
for fun1
Local
Local
Parameter
Dynamic link
Return (to main)
t
s
r
Top
ARI
for main
Local
p
at Point 1
ARI
for fun2
Local
Parameter
Dynamic link
Return (to fun1)
y
x
Top
ARI
for fun1
Local
Local
Parameter
Dynamic link
Return (to main)
t
s
r
ARI
for main
Local
p
at Point 2
ARI
for fun3
Parameter
Dynamic link
Return (to fun2)
q
Top
ARI
for fun2
Local
Parameter
Dynamic link
Return (to fun1)
y
x
ARI
for fun1
Local
Local
Parameter
Dynamic link
Return (to main)
t
s
r
ARI
for main
Local
p
at Point 3
ARI = activation record instance
N. Meng, S. Arthur
21