Activation Records

- Modern imperative programming languages typically have local variables.
  - Created upon entry to function.
  - Destroyed when function returns.
- Each invocation of a function has its own instantiation of local variables.
  - Recursive calls to a function require several instantiations to exist simultaneously.
  - Functions return only after all functions it calls have returned ⇒ last-in-first-out (LIFO) behavior.
  - A LIFO structure called a stack is used to hold each instantiation.
- The portion of the stack used for an invocation of a function is called the function’s stack frame or activation record.

The Stack

- Used to hold local variables.
- Large array which typically grows downwards in memory toward lower addresses, shrinks upwards.
- Push(rl):
  ```
  stack_pointer--; 
  M[stack_pointer] = rl;
  ```
- rl = Pop():
  ```
  rl = M[stack_pointer];
  stack_pointer++;
  ```
- Previous activation records need to be accessed, so push/pop not sufficient.
  - Treat stack as array with index off of stack_pointer.
  - Push and pop entire activation records.
Consider:

```
let
    function g(x:int) =
        let
            var y := 10
        in
            x + y
        end
    function h(y:int):int =
        y + g(y)
    in
        h(4)
end
```

Example

**Step 1: h(4) called**

Chunk of memory allocated on the stack in order to hold local variables of $h$. The activation record (or stack frame) of $h$ is pushed onto the stack.

```
Stack
Frame for h

<table>
<thead>
<tr>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
```

**Step 2: g(4) called**

Activation record for $g$ allocated (pushed) on stack.

```
Stack
Frame for h

<table>
<thead>
<tr>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Stack
Frame for g

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
```

Example

**Step 3: g(4) returns with value 14**

Activation record for $g$ deallocated (popped) from stack.

```
Stack
Frame for h

<table>
<thead>
<tr>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rv</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
</tr>
</tbody>
</table>
```

**Step 4: h(4) returns with value 18**

Activation record for $h$ deallocated (popped) from stack. Stack now empty.
Can have multiple stack frames for same function (different invocations) on stack at any given time due to recursion.

**Consider:**

```java
let
  function fact(n:int):int =
    if n = 0 then 1
    else n * fact(n - 1)
in
  fact(3)
end
```

**Step 1: Record for fact(3) pushed on stack.**

| Stack Frame for fact | n = 3 |

**Step 2: Record for fact(2) pushed on stack.**

| Stack Frame for fact | n = 3 |

| Stack Frame for fact | n = 2 |

**Step 3: Record for fact(1) pushed on stack.**

| Stack Frame for fact | n = 3 |

| Stack Frame for fact | n = 2 |

| Stack Frame for fact | n = 1 |

**Step 4: Record for fact(0) pushed on stack.**

| Stack Frame for fact | n = 3 |

| Stack Frame for fact | n = 2 |

| Stack Frame for fact | n = 1 |

| Stack Frame for fact | n = 0 |

**Step 5: Record for fact(0) popped off stack, 1 returned.**

**Step 6: Record for fact(1) popped off stack, 1 returned.**

**Step 7: Record for fact(2) popped off stack, 2 returned.**

**Step 8: Record for fact(3) popped off stack, 3 returned. Stack now empty.**
In some functional languages (such as ML, Scheme), local variables cannot be stored on stack.

```ml
fun f(x) =
  let
    fun g(y) = x + y
  in
    g
  end

Consider:
- val z = f(4)
- val w = z(5)
```

Assume variables are stack-allocated.

---

**Step 1: f(4) called.**

Frame for f(4) pushed, g returned, frame for f(4) popped.

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Frame for f</td>
</tr>
<tr>
<td>x = 4</td>
</tr>
</tbody>
</table>

Stack empty.

**Step 3: z(5) called**

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Frame for z</td>
</tr>
<tr>
<td>y = 5</td>
</tr>
</tbody>
</table>

Memory location containing x has been deallocated!

---

Combination of nested functions and functions returned as results (higher-order functions):

- Requires local variables to remain in existence even after enclosing function has been returned.
- Activation records must be allocated on heap, not stack.

Concentrate on languages which use stack.
Stack Frame Organizations

How is data organized in stack frame?

- Compiler can use any layout scheme that is convenient.
- Microprocessor manufacturers specify “standard” layout schemes used by all compilers.
  - Sometimes referred to as Calling Conventions.
  - If all compilers use the same calling conventions, then functions compiled with one compiler can call functions compiled with another.
  - Essential for interaction with OS/libraries.

Typical Stack Frame

Stack Frame Example

Suppose \( f(a_1, a_2) \) calls \( g(b_1, b_2, b_3) \)

**Step 1:**

<table>
<thead>
<tr>
<th>Frame Pointer(FP) =&gt;</th>
<th>Previous Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2</td>
<td>a1</td>
</tr>
</tbody>
</table>

| Stack Pointer(SP) => | Garbage |

**Step 2:**

<table>
<thead>
<tr>
<th>Frame Pointer(FP) =&gt;</th>
<th>Previous Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2</td>
<td>a1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack Pointer(SP) =&gt;</th>
<th>Garbage</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td></td>
</tr>
<tr>
<td>b2</td>
<td></td>
</tr>
<tr>
<td>b3</td>
<td></td>
</tr>
</tbody>
</table>
Stack Frame Example

Suppose \( f(a_1, a_2) \) calls \( g(b_1, b_2, b_3) \)

**Step 3:**

```
+-----------------+       +-----------------+
| Previous Frame  |       | Previous Frame  |
| a2              |       | a7              |
| a1              |       | a1              |
+-----------------+       +-----------------+
| Frame for f     |       | Frame for g     |
| b1              |       | b1              |
| b2              |       | b2              |
+-----------------+       +-----------------+
| Stack Pointer   |       | Stack Pointer   |
| (SP)            |       | (SP)            |
| FP              |       | FP              |
+-----------------+       +-----------------+
| OLD FP/Dynamic  |       | OLD FP/Dynamic  |
| Link            |       | Link            |
+-----------------+       +-----------------+
| Garbage         |       | Garbage         |
```

Dynamic link (AKA Control link) points to the activation record of the caller.
- Optional if size of caller activation record is known at compile time.
- Used to restore stack pointer during return sequence.

Stack Frame Example

Suppose \( f(a_1, a_2) \) calls \( g(b_1, b_2, b_3) \), and returns.

**Step 4**

```
+-----------------+       +-----------------+
| Previous Frame  |       | Previous Frame  |
| a7              |       | a7              |
| a1              |       | a1              |
+-----------------+       +-----------------+
| Frame for f     |       | Frame for f     |
| b1              |       | b1              |
+-----------------+       +-----------------+
| Stack Pointer   |       | Stack Pointer   |
| (SP)            |       | (SP)            |
| FP              |       | FP              |
+-----------------+       +-----------------+
| OLD FP/Dynamic  |       | OLD FP/Dynamic  |
| Link            |       | Link            |
+-----------------+       +-----------------+
| Garbage         |       | Garbage         |
```

**Step 5**

```
+-----------------+       +-----------------+
| Previous Frame  |       | Previous Frame  |
| a7              |       | a7              |
| a1              |       | a1              |
+-----------------+       +-----------------+
| Frame for f     |       | Frame for f     |
| b1              |       | b1              |
+-----------------+       +-----------------+
| Stack Pointer   |       | Stack Pointer   |
| (SP)            |       | (SP)            |
| FP              |       | FP              |
+-----------------+       +-----------------+
| OLD FP/Dynamic  |       | OLD FP/Dynamic  |
| Link            |       | Link            |
+-----------------+       +-----------------+
| Garbage         |       | Garbage         |
```

Parameter Passing

\( f(a_1, a_2, \ldots, a_n) \)

- Registers are faster than memory.
- Compiler should keep values in register whenever possible.
- Modern calling convention: rather than placing \( a_1, a_2, \ldots, a_n \) on stack frame, put \( a_1, \ldots, a_{k = 4} \) in registers \( r_0, r_{p+1}, r_{p+2}, r_{p+3} \) and \( a_{k+1}, a_{k+2}, a_{k+3}, \ldots, a_n \).
- If \( r_0, r_{p+1}, r_{p+2}, r_{p+3} \) are needed for other purposes, callee function must save incoming argument(s) in stack frame.
- C language allows programmer to take address of formal parameter and guarantees that formals are located at consecutive memory addresses.
  - If address argument has address taken, then it must be written into stack frame.
  - Saving it in “saved registers” area of stack won’t make it consecutive with memory resident arguments.
  - Space must be allocated even if parameters are passed through register.
If register argument has address taken, \textit{callee} writes register into corresponding space.

\begin{tabular}{|c|c|}
\hline
Frame Pointer (FP) \rightarrow &  \\
\hline
a(0) &  \\
\hline
\ldots &  \\
\hline
a(k+1) &  \\
\hline
space for a(k) &  \\
\hline
Stack Pointer (SP) \rightarrow &  \\
\hline
& space for a(2)  \\
\hline
& space for a(1)  \\
\hline
& Garbage  \\
\hline
\end{tabular}

### Registers

**Registers hold:**
- Some Parameters
- Return Value
- Local Variables
- Intermediate results of expressions (temporaries)

**Stack Frame holds:**
- Variables passed by reference or have their address taken (&)
- Variables that are accessed by procedures nested within current one.
- Variables that are too large to fit into register file.
- Array variables (address arithmetic needed to access array elements).
- Variables whose registers are needed for a specific purpose (parameter passing)
- \textit{Spilled} registers. Too many local variables to fit into register file, so some must be stored in stack frame.

---

Compilers typically place variables on stack until it can determine whether or not it can be promoted to a register (e.g. no references).

The assignment of variables to registers is done by the \textit{Register Allocator}.
Registers

Register state for a function must be saved before a callee function can use them.

Calling convention describes two types of registers.

- **Caller-save** register are the responsibility of the calling function.
  - Caller-save register values are saved to the stack by the calling function if they will be used after the call.
  - The callee function can use caller-save registers without saving their original values.

- **Callee-save** registers are the responsibility of the called function.
  - Callee-save register values must be saved to the stack by called function before they can be used.
  - The caller (calling function) can assume that these registers will contain the same value before and after the call.

Placement of values into callee-save vs. caller-save registers is determined by the register allocator.

Return Address and Return Value

A called function must be able to return to calling function when finished.

- Return address is address of instruction following the function call.
- Return address can be placed on stack or in a register.
- The call instruction in modern machines places the return address in a designated register.
- This return address is written to stack by callee function in non-leaf functions.

Return value is placed in designated register by callee function.

Frame Resident Variables

- A variable escapes if:
  - it is passed by reference,
  - its address is taken, or
  - it is accessed from a nested function.

- Variables cannot be assigned a location at declaration time.
  - Escape conditions not known.
  - Assign provisional locations, decide later if variables can be promoted to registers.

- escape set to true by default.
In languages that allow nested functions (e.g. Tiger), functions must access outer function’s stack frame.

```plaintext
let
  function f():int = let
    var a:=5
    function g(y:int):int = let
      var b:=10
      function h(z:int):int =
        if z > 10 then h(z / 2)
        else z + b * a <<< b, a of outer fn
      in
        y + a + h(16) <<< a of outer fn
    end
  in g(10)
end
in f() end
```

**Solution:**

Whenever `f` is called, it is passed pointer to most recent activation record of `g` that immediately encloses `f` in program text ⇒ Static Link (AKA Access Link).

```
Frame Pointer(FP) <--
  x = 3
Stack Pointer(SP) <--
```

```
Frame for f
```

```
FP <--
  y = 10
```

```
Frame for g
```

```
SP <--
  &y = FP + 1
&x = M[FP] - 1
```

---

```
Frame Pointer(FP) <--
  x = 5
Stack Pointer(SP) <--
```

```
Frame for f
```

```
FP <--
  z = 16
```

```
Frame for h
```

```
SP <--
  &z = FP + 1
```

```
Frame for h
```

```
FP <--
  z = 8
```

```
Frame for h
```

```
SP <--
  &z = FP + 1
&b = M[FP] - 2
&x = M[M[FP]] - 1
```
• Need a chain of indirect memory references for each variable access.
• Number of indirect references = difference in nesting depth between variable declaration function and use function.