Function calls and Stack

Madhavan Mukund, **S P Suresh**

Programming Language Concepts
Lecture 7
31 January 2023

General layout of a program in memory

Activation record

- Contains information pertaining to a function invocation

Activation record

- Contains information pertaining to a function invocation
- Added to the top of the stack at the start of the function invocation

Activation record

- Contains information pertaining to a function invocation
- Added to the top of the stack at the start of the function invocation
- Removed from the stack at the end of the function invocation

Activation record

- Contains information pertaining to a function invocation
- Added to the top of the stack at the start of the function invocation
- Removed from the stack at the end of the function invocation
- Stores parameters, local variables, temporary variables used in running the function

Activation record

- Contains information pertaining to a function invocation
- Added to the top of the stack at the start of the function invocation
- Removed from the stack at the end of the function invocation
- Stores parameters, local variables, temporary variables used in running the function
- Various pointers – **Control link**, **access link**, **return address**

Activation record

- Contains information pertaining to a function invocation
- Added to the top of the stack at the start of the function invocation
- Removed from the stack at the end of the function invocation
- Stores parameters, local variables, temporary variables used in running the function
- Various pointers – **Control link**, **access link**, **return address**
- **System-wide pointers**

Activation record

- Contains information pertaining to a function invocation
- Added to the top of the stack at the start of the function invocation
- Removed from the stack at the end of the function invocation
- Stores parameters, local variables, temporary variables used in running the function
- Various pointers – **Control link**, **access link**, **return address**
- **System-wide pointers**
  - **Program counter** – address of the next instruction to execute

Activation record

- Contains information pertaining to a function invocation
- Added to the top of the stack at the start of the function invocation
- Removed from the stack at the end of the function invocation
- Stores parameters, local variables, temporary variables used in running the function
- Various pointers – **Control link**, **access link**, **return address**
- **System-wide pointers**
  - **Program counter** – address of the next instruction to execute
  - **Stack pointer** – points to the top of the system stack

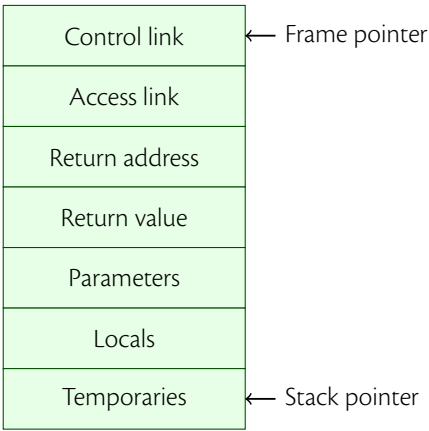
Activation record

- Contains information pertaining to a function invocation
- Added to the top of the stack at the start of the function invocation
- Removed from the stack at the end of the function invocation
- Stores parameters, local variables, temporary variables used in running the function
- Various pointers – **Control link**, **access link**, **return address**
- **System-wide pointers**
  - **Program counter** – address of the next instruction to execute
  - **Stack pointer** – points to the top of the system stack
  - **Frame pointer** – points to the start of the topmost frame on stack

Activation record

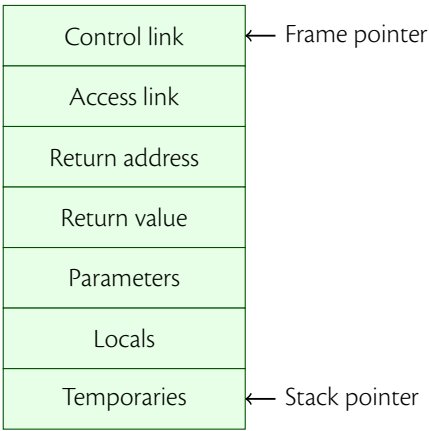
- Contains information pertaining to a function invocation
- Added to the top of the stack at the start of the function invocation
- Removed from the stack at the end of the function invocation
- Stores parameters, local variables, temporary variables used in running the function
- Various pointers – **Control link**, **access link**, **return address**
- **System-wide pointers**
  - **Program counter** – address of the next instruction to execute
  - **Stack pointer** – points to the top of the system stack
  - **Frame pointer** – points to the start of the topmost frame on stack
  - Data in topmost frame accessed via **offsets** from the frame pointer or stack pointer – offsets can computed at **compile time**

Activation record ...



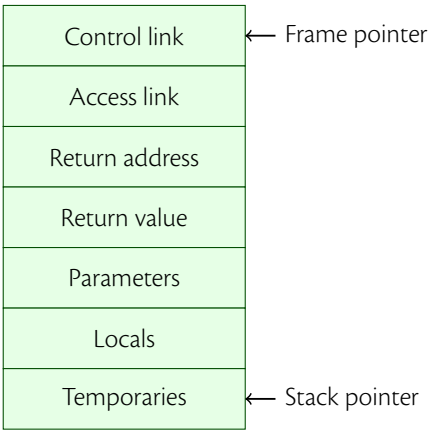
- **Control link** points to activation record of caller

Activation record ...



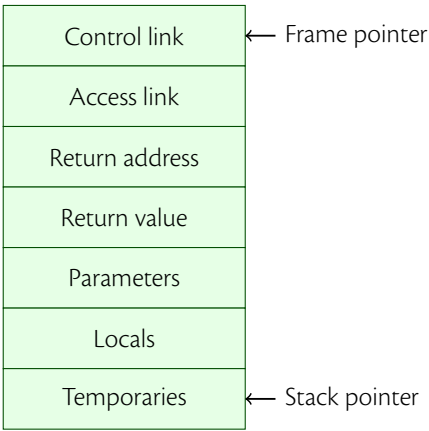
- **Control link** points to activation record of caller
- **Access link** is for non-local variable access

Activation record ...



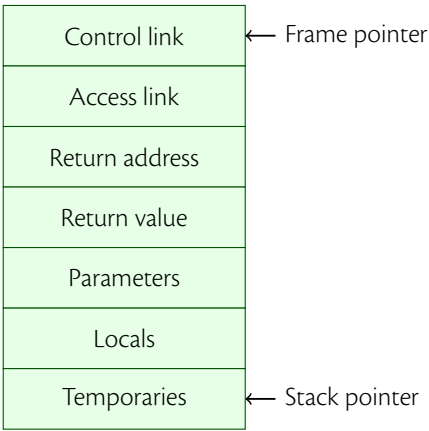
- **Control link** points to activation record of caller
- **Access link** is for non-local variable access
- **Return address** is the address of first instruction to execute after the function call returns

Activation record ...



- **Control link** points to activation record of caller
- **Access link** is for non-local variable access
- **Return address** is the address of first instruction to execute after the function call returns
- **Return value** stores the return value, which should be picked up by the caller

Activation record ...



- **Control link** points to activation record of caller
- **Access link** is for non-local variable access
- **Return address** is the address of first instruction to execute after the function call returns
- **Return value** stores the return value, which should be picked up by the caller
- **Temporaries** are locations to store intermediate values in

Function calls

- On a new function invocation, create a new frame

Function calls

- On a new function invocation, create a new frame
- The start of the frame is the control link, so we do the following updates

Function calls

- On a new function invocation, create a new frame
- The start of the frame is the control link, so we do the following updates
- `Stack[SP+1] = FP; SP += 1; FP = SP;`

Function calls

- On a new function invocation, create a new frame
- The start of the frame is the control link, so we do the following updates
- `Stack[SP+1] = FP; SP += 1; FP = SP;`
- Update the **program counter (PC)** to point to the start of the callee

Function calls

- On a new function invocation, create a new frame
- The start of the frame is the control link, so we do the following updates
- `Stack[SP+1] = FP; SP += 1; FP = SP;`
- Update the **program counter (PC)** to point to the start of the callee
- Store the next instruction in the caller in the return address just before changing the PC

Function calls

- On a new function invocation, create a new frame
- The start of the frame is the control link, so we do the following updates
- `Stack[SP+1] = FP; SP += 1; FP = SP;`
- Update the **program counter (PC)** to point to the start of the callee
- Store the next instruction in the caller in the return address just before changing the PC
- Store parameters, locals, etc. on stack …SP will change but not FP!

Function returns

- At the end of a function call, set `SP = FP - 1`

Function returns

- At the end of a function call, set `SP = FP - 1`
- Set `PC = Stack[FP+2]` (this is the return address)

Function returns

- At the end of a function call, set `SP = FP - 1`
- Set `PC = Stack[FP+2]` (this is the return address)
- Set `FP = Stack[FP]` (this contains the control link)

Function returns

- At the end of a function call, set `SP = FP - 1`
- Set `PC = Stack[FP+2]` (this is the return address)
- Set `FP = Stack[FP]` (this contains the control link)
- Set `Stack[FP+offset] = Stack[SP+4]` (copy the return value to the local variable that expects the value)

Function returns

- At the end of a function call, set `SP = FP - 1`
- Set `PC = Stack[FP+2]` (this is the return address)
- Set `FP = Stack[FP]` (this contains the control link)
- Set `Stack[FP+offset] = Stack[SP+4]` (copy the return value to the local variable that expects the value)
- The stack frame has been deallocated

Function returns

- At the end of a function call, set `SP = FP - 1`
- Set `PC = Stack[FP+2]` (this is the return address)
- Set `FP = Stack[FP]` (this contains the control link)
- Set `Stack[FP+offset] = Stack[SP+4]` (copy the return value to the local variable that expects the value)
- The stack frame has been deallocated
- The caller is now the topmost frame

Function returns

- At the end of a function call, set `SP = FP - 1`
- Set `PC = Stack[FP+2]` (this is the return address)
- Set `FP = Stack[FP]` (this contains the control link)
- Set `Stack[FP+offset] = Stack[SP+4]` (copy the return value to the local variable that expects the value)
- The stack frame has been deallocated
- The caller is now the topmost frame
- If another function is called, we overwrite the locations occupied by the previous callee

Access links

```
func f {
    int x = 0;
    int fib(int n) {
        if n ⩽ 1 then return n;
        else {
            x += 1;
            return fib(n-1) + fib(n-2);
        }
    }
    print(fib(4));
}
```

- Count the number of additions in `fib(4)`

Access links

```
func f {
    int x = 0;
    int fib(int n) {
        if n ⩽ 1 then return n;
        else {
            x += 1;
            return fib(n-1) + fib(n-2);
        }
    }
    print(fib(4));
}
```

- Count the number of additions in `fib(4)`
- `x` is non-local

Access links

```
func f {
    int x = 0;
    int fib(int n) {
        if n ⩽ 1 then return n;
        else {
            x += 1;
            return fib(n-1) + fib(n-2);
        }
    }
    print(fib(4));
}
```

- Count the number of additions in `fib(4)`
- `x` is non-local
- `fib(4)` is called by `f`, so `x` can be accessed by following the control link

Access links

```
func f {
    int x = 0;
    int fib(int n) {
        if n ⩽ 1 then return n;
        else {
            x += 1;
            return fib(n-1) + fib(n-2);
        }
    }
    print(fib(4));
}
```

- Count the number of additions in `fib(4)`
- `x` is non-local
- `fib(4)` is called by `f`, so `x` can be accessed by following the control link
- But `fib(3)` is called by `fib(4)`, so control link cannot be used to access `x`

Access links

```
func f {
    int x = 0;
    int fib(int n) {
        if n ⩽ 1 then return n;
        else {
            x += 1;
            return fib(n-1) + fib(n-2);
        }
    }
    print(fib(4));
}
```

- Count the number of additions in `fib(4)`
- `x` is non-local
- `fib(4)` is called by `f`, so `x` can be accessed by following the control link
- But `fib(3)` is called by `fib(4)`, so control link cannot be used to access `x`
- Need a new kind of link – **access link**

Setting access links

- This is needed for functions/classes defined inside the scope of other functions/classes

Setting access links

- This is needed for functions/classes defined inside the scope of other functions/classes
- Say functions `g` and `h` are defined inside function `f`

Setting access links

- This is needed for functions/classes defined inside the scope of other functions/classes
- Say functions `g` and `h` are defined inside function `f`
- Say `x` is a variable defined inside `f`

Setting access links

- This is needed for functions/classes defined inside the scope of other functions/classes
- Say functions `g` and `h` are defined inside function `f`
- Say `x` is a variable defined inside `f`
- `g` and `h` are recursive and can call each other

Setting access links

- This is needed for functions/classes defined inside the scope of other functions/classes
- Say functions `g` and `h` are defined inside function `f`
- Say `x` is a variable defined inside `f`
- `g` and `h` are recursive and can call each other
- The body of `f` can call `g` or `h`

Setting access links

- This is needed for functions/classes defined inside the scope of other functions/classes
- Say functions `g` and `h` are defined inside function `f`
- Say `x` is a variable defined inside `f`
- `g` and `h` are recursive and can call each other
- The body of `f` can call `g` or `h`
- We cannot call them outside the scope of `f`

- This is needed for functions/classes defined inside the scope of other functions/classes
- Say functions `g` and `h` are defined inside function `f`
- Say `x` is a variable defined inside `f`
- `g` and `h` are recursive and can call each other
- The body of `f` can call `g` or `h`
- We cannot call them outside the scope of `f`

Setting access links

- Access link of each recursive call of `g` and `h` should point to the frame of `f`

- This is needed for functions/classes defined inside the scope of other functions/classes
- Say functions g and h are defined inside function f
- Say x is a variable defined inside f
- g and h are recursive and can call each other
- The body of f can call g or h
- We cannot call them outside the scope of f

Setting access links

- Access link of each recursive call of g and h should point to the frame of f
- Any time g or h is called, f is already on the call stack

- This is needed for functions/classes defined inside the scope of other functions/classes
- Say functions `g` and `h` are defined inside function `f`
- Say `x` is a variable defined inside `f`
- `g` and `h` are recursive and can call each other
- The body of `f` can call `g` or `h`
- We cannot call them outside the scope of `f`

Setting access links

- Access link of each recursive call of `g` and `h` should point to the frame of `f`
- Any time `g` or `h` is called, `f` is already on the call stack
- When the call to `g` is made from `f`, set the access link to be the same as the control link

Setting access links

- This is needed for functions/classes defined inside the scope of other functions/classes
- Say functions `g` and `h` are defined inside function `f`
- Say `x` is a variable defined inside `f`
- `g` and `h` are recursive and can call each other
- The body of `f` can call `g` or `h`
- We cannot call them outside the scope of `f`

- Access link of each recursive call of `g` and `h` should point to the frame of `f`
- Any time `g` or `h` is called, `f` is already on the call stack
- When the call to `g` is made from `f`, set the access link to be the same as the control link
- When there is a call from `g`/`h` to `g`/`h`, set the access link of the callee to be the same as the access link of the caller!

Chains of access links

- Once can have nested function/class definitions!

Chains of access links

- Once can have nested function/class definitions!
- h defined inside g, which is defined inside f

Chains of access links

- Once can have nested function/class definitions!
- `h` defined inside `g`, which is defined inside `f`
- Possible for `h` to access variables defined innside `f`

Chains of access links

- Once can have nested function/class definitions!
- h defined inside g, which is defined inside f
- Possible for h to access variables defined innside f
- Access link of h points to g, that of g points to f

Chains of access links

- Once can have nested function/class definitions!
- h defined inside g, which is defined inside f
- Possible for h to access variables defined innside f
- Access link of h points to g, that of g points to f
- We might have to follow a **chain of access links** to access non-local variables

Chains of access links

- Once can have nested function/class definitions!
- h defined inside g, which is defined inside f
- Possible for h to access variables defined innside f
- Access link of h points to g, that of g points to f
- We might have to follow a **chain of access links** to access non-local variables
- The number of hops to follow in this chain can be determined at **compile time**!

Chains of access links

- Once can have nested function/class definitions!
- h defined inside g, which is defined inside f
- Possible for h to access variables defined innside f
- Access link of h points to g, that of g points to f
- We might have to follow a **chain of access links** to access non-local variables
- The number of hops to follow in this chain can be determined at **compile time**!
- Suppose x is declared in f, and y in g

Chains of access links

- Once can have nested function/class definitions!
- h defined inside g, which is defined inside f
- Possible for h to access variables defined innside f
- Access link of h points to g, that of g points to f
- We might have to follow a **chain of access links** to access non-local variables
- The number of hops to follow in this chain can be determined at **compile time**!
- Suppose x is declared in f, and y in g
- To access y in h we go to the frame pointed to by access link

Chains of access links

- Once can have nested function/class definitions!
- h defined inside g, which is defined inside f
- Possible for h to access variables defined innside f
- Access link of h points to g, that of g points to f
- We might have to follow a **chain of access links** to access non-local variables
- The number of hops to follow in this chain can be determined at **compile time**!
- Suppose x is declared in f, and y in g
- To access y in h we go to the frame pointed to by access link
- To access x in h we go to the frame pointed to by access link, go to the frame pointed to by the access link **there**, and access x!