Lecture 24-26

Runtime Environment

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

April 4, 2025

Runtime Stack

```
#include <stdio.h>
int *p;
int val;
void foo(int a){
    p = &a;
    val = *(--p) ;
    printf("In foo\n");
}
void bar(int b){
    p = &b ;
    *(--p) = val ;
    printf("In bar\n");
}
int main(){
    int a = 1, b = 2;
    foo(a);
    bar(b);
    return 0;
}
```

Support required for a function call

Support required for a function call

- `jump` foo (why so hype about this)

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo
  - Store in a global variable

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo
  - Store in a global variable
  - store in Register

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo
  - Store in a global variable
  - store in Register
- How to check stack is going upwards or downwards

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo
  - Store in a global variable
  - store in Register
- How to check stack is going upwards or downwards
- Parameters

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo
  - Store in a global variable
  - store in Register
- How to check stack is going upwards or downwards
- Parameters
- Does order matter?

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo
  - Store in a global variable
  - store in Register
- How to check stack is going upwards or downwards
- Parameters
- Does order matter?
- Context

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo
  - Store in a global variable
  - store in Register
- How to check stack is going upwards or downwards
- Parameters
- Does order matter?
- Context
- Local variable

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo
  - Store in a global variable
  - store in Register
- How to check stack is going upwards or downwards
- Parameters
- Does order matter?
- Context
- Local variable
- Can we store local variables in registers?

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo
  - Store in a global variable
  - store in Register
- How to check stack is going upwards or downwards
- Parameters
- Does order matter?
- Context
- Local variable
- Can we store local variables in registers?
- Frame pointer

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo
  - Store in a global variable
  - store in Register
- How to check stack is going upwards or downwards
- Parameters
- Does order matter?
- Context
- Local variable
- Can we store local variables in registers?
- Frame pointer
  - Why can't the compiler modify the address when it is updating the esp?

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo
  - Store in a global variable
  - store in Register
- How to check stack is going upwards or downwards
- Parameters
- Does order matter?
- Context
- Local variable
- Can we store local variables in registers?
- Frame pointer
  - Why can't the compiler modify the address when it is updating the esp?
  - Actual use is during debugging

Support required for a function call

- `jump` foo (why so hype about this)
- Return value and return address
  - store return address in the caller and do *jumpBack* foo
  - Store in a global variable
  - store in Register
- How to check stack is going upwards or downwards
- Parameters
- Does order matter?
- Context
- Local variable
- Can we store local variables in registers?
- Frame pointer
  - Why can't the compiler modify the address when it is updating the esp?
  - Actual use is during debugging
  - Can always generate with esp using the flag "`-fomit-frame-pointer`"

Runtime Environment

- Relationship between names and data objects (of target machine)

Runtime Environment

- Relationship between names and data objects (of target machine)
- Allocation and de-allocation is managed by run time support package

Runtime Environment

- Relationship between names and data objects (of target machine)
- Allocation and de-allocation is managed by run time support package
- Each execution of a procedure is an activation of the procedure. If procedure is recursive, several activations may be alive at the same time.

Runtime Environment

- Relationship between names and data objects (of target machine)
- Allocation and de-allocation is managed by run time support package
- Each execution of a procedure is an activation of the procedure. If procedure is recursive, several activations may be alive at the same time.
  - If a and b are activations of two procedures then their lifetime is either non overlapping or nested

Runtime Environment

- Relationship between names and data objects (of target machine)
- Allocation and de-allocation is managed by run time support package
- Each execution of a procedure is an activation of the procedure. If procedure is recursive, several activations may be alive at the same time.
  - If a and b are activations of two procedures then their lifetime is either non overlapping or nested
  - A procedure is recursive if an activation can begin before an earlier activation of the same procedure has ended

Procedure

- A procedure definition is a declaration that associates an identifier with a statement (procedure body)

Procedure

- A procedure definition is a declaration that associates an identifier with a statement (procedure body)
- When a procedure name appears in an executable statement, it is called at that point

Procedure

- A procedure definition is a declaration that associates an identifier with a statement (procedure body)
- When a procedure name appears in an executable statement, it is called at that point
- Formal parameters are the one that appear in declaration. Actual Parameters are the one that appear in when a procedure is called

Activation Tree

- Control flows sequentially

Activation Tree

- Control flows sequentially
- Execution of a procedure starts at the beginning of body

Activation Tree

- Control flows sequentially
- Execution of a procedure starts at the beginning of body
- It returns control to place where procedure was called from

Activation Tree

- Control flows sequentially
- Execution of a procedure starts at the beginning of body
- It returns control to place where procedure was called from
- A tree can be used, called an activation tree, to depict the way control enters and leaves activations

Activation Tree

- Control flows sequentially
- Execution of a procedure starts at the beginning of body
- It returns control to place where procedure was called from
- A tree can be used, called an activation tree, to depict the way control enters and leaves activations
    - The root represents the activation of main program

Activation Tree

- Control flows sequentially
- Execution of a procedure starts at the beginning of body
- It returns control to place where procedure was called from
- A tree can be used, called an activation tree, to depict the way control enters and leaves activations
  - The root represents the activation of main program
  - Each node represents an activation of procedure

Activation Tree

- Control flows sequentially
- Execution of a procedure starts at the beginning of body
- It returns control to place where procedure was called from
- A tree can be used, called an activation tree, to depict the way control enters and leaves activations
  - The root represents the activation of main program
  - Each node represents an activation of procedure
  - The node a is parent of b if control flows from a to b

Activation Tree

- Control flows sequentially
- Execution of a procedure starts at the beginning of body
- It returns control to place where procedure was called from
- A tree can be used, called an activation tree, to depict the way control enters and leaves activations
  - The root represents the activation of main program
  - Each node represents an activation of procedure
  - The node a is parent of b if control flows from a to b
  - The node a is to the left of node b if lifetime of a occurs before b

Control Stack

- Flow of control in program corresponds to depth first traversal of activation tree

Control Stack

- Flow of control in program corresponds to depth first traversal of activation tree
- Use a stack called control stack to keep track of live procedure activations

Control Stack

- Flow of control in program corresponds to depth first traversal of activation tree
- Use a stack called control stack to keep track of live procedure activations
- Push the node when activation begins and pop the node when activation ends

Control Stack

- Flow of control in program corresponds to depth first traversal of activation tree
- Use a stack called control stack to keep track of live procedure activations
- Push the node when activation begins and pop the node when activation ends
- When the node n is at the top of the stack the stack contains the nodes along the path from n to the root

Storage Organization

- The runtime storage might be subdivided into

Storage Organization

- The runtime storage might be subdivided into
  - Target code

Storage Organization

- The runtime storage might be subdivided into
  - Target code
  - Data objects

Storage Organization

- The runtime storage might be subdivided into
  - Target code
  - Data objects
  - Stack to keep track of procedure activation

Storage Organization

- The runtime storage might be subdivided into
  - Target code
  - Data objects
  - Stack to keep track of procedure activation
  - Heap to keep all other information

Activation Record

- temporaries: used in expression evaluation

Activation Record

- temporaries: used in expression evaluation
- local data: field for local data

Activation Record

- temporaries: used in expression evaluation
- local data: field for local data
- saved machine status: holds info about machine status before procedure call

Activation Record

- temporaries: used in expression evaluation
- local data: field for local data
- saved machine status: holds info about machine status before procedure call
- access link : to access non local data

Activation Record

- temporaries: used in expression evaluation
- local data: field for local data
- saved machine status: holds info about machine status before procedure call
- access link : to access non local data
- control link :points to activation record of caller

Activation Record

- temporaries: used in expression evaluation
- local data: field for local data
- saved machine status: holds info about machine status before procedure call
- access link : to access non local data
- control link :points to activation record of caller
- actual parameters: field to hold actual parameters

Activation Record

- temporaries: used in expression evaluation
- local data: field for local data
- saved machine status: holds info about machine status before procedure call
- access link : to access non local data
- control link :points to activation record of caller
- actual parameters: field to hold actual parameters
- returned value: field for holding value to be returned

Issues to be addressed

- Can procedures be recursive?

Issues to be addressed

- Can procedures be recursive?
- What happens to locals when procedures return from an activation?

Issues to be addressed

- Can procedures be recursive?
- What happens to locals when procedures return from an activation?
- Can procedure refer to non local names?

Issues to be addressed

- Can procedures be recursive?
- What happens to locals when procedures return from an activation?
- Can procedure refer to non local names?
- How to pass parameters?

Issues to be addressed

- Can procedures be recursive?
- What happens to locals when procedures return from an activation?
- Can procedure refer to non local names?
- How to pass parameters?
- Can procedure be parameter?

Issues to be addressed

- Can procedures be recursive?
- What happens to locals when procedures return from an activation?
- Can procedure refer to non local names?
- How to pass parameters?
- Can procedure be parameter?
- Can procedure be returned?

Issues to be addressed

- Can procedures be recursive?
- What happens to locals when procedures return from an activation?
- Can procedure refer to non local names?
- How to pass parameters?
- Can procedure be parameter?
- Can procedure be returned?
- Can storage be dynamically allocated?

Issues to be addressed

- Can procedures be recursive?
- What happens to locals when procedures return from an activation?
- Can procedure refer to non local names?
- How to pass parameters?
- Can procedure be parameter?
- Can procedure be returned?
- Can storage be dynamically allocated?
- Can storage be de-allocated?

Layout of local data

- Assume byte is the smallest unit

Layout of local data

- Assume byte is the smallest unit
- Multi-byte objects are stored in consecutive bytes and given address of first byte

Layout of local data

- Assume byte is the smallest unit
- Multi-byte objects are stored in consecutive bytes and given address of first byte
- The amount of storage needed is determined by its type

Layout of local data

- Assume byte is the smallest unit
- Multi-byte objects are stored in consecutive bytes and given address of first byte
- The amount of storage needed is determined by its type
- Memory allocation is done as the declarations are processed

Layout of local data

- Assume byte is the smallest unit
- Multi-byte objects are stored in consecutive bytes and given address of first byte
- The amount of storage needed is determined by its type
- Memory allocation is done as the declarations are processed
- Data may have to be aligned (in a word) padding is done to have alignment.

Storage Allocation Strategies

- Static allocation: lays out storage at compile time for all data objects

Storage Allocation Strategies

- Static allocation: lays out storage at compile time for all data objects
- Stack allocation: manages the runtime storage as a stack

Storage Allocation Strategies

- Static allocation: lays out storage at compile time for all data objects
- Stack allocation: manages the runtime storage as a stack
- Heap allocation: allocates and de- allocates storage as needed at runtime from heap

Static allocation

- Names are bound to storage as the program is compiled

Static allocation

- Names are bound to storage as the program is compiled
- No runtime support is required

Static allocation

- Names are bound to storage as the program is compiled
- No runtime support is required
- Bindings do not change at run time

Static allocation

- Names are bound to storage as the program is compiled
- No runtime support is required
- Bindings do not change at run time
- On every invocation of procedure names are bound to the same storage

Static allocation

- Names are bound to storage as the program is compiled
- No runtime support is required
- Bindings do not change at run time
- On every invocation of procedure names are bound to the same storage
- Values of local names are retained across activations of a procedure

Static allocation

- Names are bound to storage as the program is compiled
- No runtime support is required
- Bindings do not change at run time
- On every invocation of procedure names are bound to the same storage
- Values of local names are retained across activations of a procedure
- Type of a name determines the amount of storage to be set aside

Static allocation

- Names are bound to storage as the program is compiled
- No runtime support is required
- Bindings do not change at run time
- On every invocation of procedure names are bound to the same storage
- Values of local names are retained across activations of a procedure
- Type of a name determines the amount of storage to be set aside
- Address of a storage consists of an offset from the end of an activation record

Static allocation

- Names are bound to storage as the program is compiled
- No runtime support is required
- Bindings do not change at run time
- On every invocation of procedure names are bound to the same storage
- Values of local names are retained across activations of a procedure
- Type of a name determines the amount of storage to be set aside
- Address of a storage consists of an offset from the end of an activation record
- All the addresses can be filled at compile time

Static allocation

- Names are bound to storage as the program is compiled
- No runtime support is required
- Bindings do not change at run time
- On every invocation of procedure names are bound to the same storage
- Values of local names are retained across activations of a procedure
- Type of a name determines the amount of storage to be set aside
- Address of a storage consists of an offset from the end of an activation record
- All the addresses can be filled at compile time
    - Size of all data objects must be known at compile time

Static allocation

- Names are bound to storage as the program is compiled
- No runtime support is required
- Bindings do not change at run time
- On every invocation of procedure names are bound to the same storage
- Values of local names are retained across activations of a procedure
- Type of a name determines the amount of storage to be set aside
- Address of a storage consists of an offset from the end of an activation record
- All the addresses can be filled at compile time
  - Size of all data objects must be known at compile time
  - Data structures cannot be created dynamically

Stack Allocation

- Calling Sequence

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field
  - A return sequence restores the state of the machine so that calling procedure can continue execution

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field
  - A return sequence restores the state of the machine so that calling procedure can continue execution
- Call Sequence

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field
  - A return sequence restores the state of the machine so that calling procedure can continue execution
- Call Sequence
  - Caller evaluates the actual parameters

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field
  - A return sequence restores the state of the machine so that calling procedure can continue execution
- Call Sequence
  - Caller evaluates the actual parameters
  - Caller stores return address and other values (control link) into callee's activation record

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field
  - A return sequence restores the state of the machine so that calling procedure can continue execution
- Call Sequence
  - Caller evaluates the actual parameters
  - Caller stores return address and other values (control link) into callee's activation record
  - Callee saves register values and other status information

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field
  - A return sequence restores the state of the machine so that calling procedure can continue execution
- Call Sequence
  - Caller evaluates the actual parameters
  - Caller stores return address and other values (control link) into callee's activation record
  - Callee saves register values and other status information
  - Callee initializes its local data and begins execution

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field
  - A return sequence restores the state of the machine so that calling procedure can continue execution
- Call Sequence
  - Caller evaluates the actual parameters
  - Caller stores return address and other values (control link) into callee's activation record
  - Callee saves register values and other status information
  - Callee initializes its local data and begins execution
- Return Sequence

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field
  - A return sequence restores the state of the machine so that calling procedure can continue execution
- Call Sequence
  - Caller evaluates the actual parameters
  - Caller stores return address and other values (control link) into callee's activation record
  - Callee saves register values and other status information
  - Callee initializes its local data and begins execution
- Return Sequence
  - Callee places a return value next to activation record of caller

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field
  - A return sequence restores the state of the machine so that calling procedure can continue execution
- Call Sequence
  - Caller evaluates the actual parameters
  - Caller stores return address and other values (control link) into callee's activation record
  - Callee saves register values and other status information
  - Callee initializes its local data and begins execution
- Return Sequence
  - Callee places a return value next to activation record of caller
  - Restores registers using information in status field

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field
  - A return sequence restores the state of the machine so that calling procedure can continue execution
- Call Sequence
  - Caller evaluates the actual parameters
  - Caller stores return address and other values (control link) into callee's activation record
  - Callee saves register values and other status information
  - Callee initializes its local data and begins execution
- Return Sequence
  - Callee places a return value next to activation record of caller
  - Restores registers using information in status field
  - Branch to return address

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field
  - A return sequence restores the state of the machine so that calling procedure can continue execution
- Call Sequence
  - Caller evaluates the actual parameters
  - Caller stores return address and other values (control link) into callee's activation record
  - Callee saves register values and other status information
  - Callee initializes its local data and begins execution
- Return Sequence
  - Callee places a return value next to activation record of caller
  - Restores registers using information in status field
  - Branch to return address
  - Caller copies return value into its own activation record

Stack Allocation

- Calling Sequence
  - A call sequence allocates an activation record and enters information into its field
  - A return sequence restores the state of the machine so that calling procedure can continue execution
- Call Sequence
  - Caller evaluates the actual parameters
  - Caller stores return address and other values (control link) into callee's activation record
  - Callee saves register values and other status information
  - Callee initializes its local data and begins execution
- Return Sequence
  - Callee places a return value next to activation record of caller
  - Restores registers using information in status field
  - Branch to return address
  - Caller copies return value into its own activation record
- Long/variable length data

Heap Allocation

- Stack allocation cannot be used if:

Heap Allocation

- Stack allocation cannot be used if:
  - The values of the local variables must be retained when an activation ends

Heap Allocation

- Stack allocation cannot be used if:
  - The values of the local variables must be retained when an activation ends
  - A called activation outlives the caller

Heap Allocation

- Stack allocation cannot be used if:
  - The values of the local variables must be retained when an activation ends
  - A called activation outlives the caller
- De-allocation of activation record cannot occur in last-in first-out fashion

Heap Allocation

- Stack allocation cannot be used if:
  - The values of the local variables must be retained when an activation ends
  - A called activation outlives the caller
- De-allocation of activation record cannot occur in last-in first-out fashion
- Heap allocation gives out pieces of contiguous storage for activation records.

Heap Allocation

- Stack allocation cannot be used if:
  - The values of the local variables must be retained when an activation ends
  - A called activation outlives the caller
- De-allocation of activation record cannot occur in last-in first-out fashion
- Heap allocation gives out pieces of contiguous storage for activation records.
- Pieces may be de-allocated in any order.

Heap Allocation

- Stack allocation cannot be used if:
  - The values of the local variables must be retained when an activation ends
  - A called activation outlives the caller
- De-allocation of activation record cannot occur in last-in first-out fashion
- Heap allocation gives out pieces of contiguous storage for activation records.
- Pieces may be de-allocated in any order.
- Over time the heap will consist of alternate areas that are free and in use

Heap Allocation

- Stack allocation cannot be used if:
  - The values of the local variables must be retained when an activation ends
  - A called activation outlives the caller
- De-allocation of activation record cannot occur in last-in first-out fashion
- Heap allocation gives out pieces of contiguous storage for activation records.
- Pieces may be de-allocated in any order.
- Over time the heap will consist of alternate areas that are free and in use
- Heap manager is supposed to make use of the free space

Heap Allocation

- Stack allocation cannot be used if:
  - The values of the local variables must be retained when an activation ends
  - A called activation outlives the caller
- De-allocation of activation record cannot occur in last-in first-out fashion
- Heap allocation gives out pieces of contiguous storage for activation records.
- Pieces may be de-allocated in any order.
- Over time the heap will consist of alternate areas that are free and in use
- Heap manager is supposed to make use of the free space
- For each size of interest keep a linked list of free blocks of that size

Access to non-local names

- Scope rules determine the treatment of non-local names

Access to non-local names

- Scope rules determine the treatment of non-local names
- A common rule is lexical scoping or static scoping (most languages use lexical scoping)

Access to non-local names

- Scope rules determine the treatment of non-local names
- A common rule is lexical scoping or static scoping (most languages use lexical scoping)
- Blocks

Access to non-local names

- Scope rules determine the treatment of non-local names
- A common rule is lexical scoping or static scoping (most languages use lexical scoping)
- Blocks
  - A block statement contains its own data declarations

Access to non-local names

- Scope rules determine the treatment of non-local names
- A common rule is lexical scoping or static scoping (most languages use lexical scoping)
- Blocks
  - A block statement contains its own data declarations
  - Blocks can be nested

Access to non-local names

- Scope rules determine the treatment of non-local names
- A common rule is lexical scoping or static scoping (most languages use lexical scoping)
- Blocks
  - A block statement contains its own data declarations
  - Blocks can be nested
- Scope of the declaration is given by most closely nested rule

Access to non-local names

- Scope rules determine the treatment of non-local names
- A common rule is lexical scoping or static scoping (most languages use lexical scoping)
- Blocks
  - A block statement contains its own data declarations
  - Blocks can be nested
- Scope of the declaration is given by most closely nested rule
  - The scope of a declaration in block B includes B

Access to non-local names

- Scope rules determine the treatment of non-local names
- A common rule is lexical scoping or static scoping (most languages use lexical scoping)
- Blocks
  - A block statement contains its own data declarations
  - Blocks can be nested
- Scope of the declaration is given by most closely nested rule
  - The scope of a declaration in block B includes B
  - If a name X is not declared in B then an occurrence of X is in the scope of declarator X in B such that

Access to non-local names

- Scope rules determine the treatment of non-local names
- A common rule is lexical scoping or static scoping (most languages use lexical scoping)
- Blocks
  - A block statement contains its own data declarations
  - Blocks can be nested
- Scope of the declaration is given by most closely nested rule
  - The scope of a declaration in block B includes B
  - If a name X is not declared in B then an occurrence of X is in the scope of declarator X in B such that
    - B has a declaration of X

Access to non-local names

- Scope rules determine the treatment of non-local names
- A common rule is lexical scoping or static scoping (most languages use lexical scoping)
- Blocks
  - A block statement contains its own data declarations
  - Blocks can be nested
- Scope of the declaration is given by most closely nested rule
  - The scope of a declaration in block B includes B
  - If a name X is not declared in B then an occurrence of X is in the scope of declarator X in B such that
    - B has a declaration of X
    - B is most closely nested around B

Lexical scope without nested procedures

- A procedure definition cannot occur within another

Lexical scope without nested procedures

- A procedure definition cannot occur within another
- Therefore, all non local references are global and can be allocated at compile time

Lexical scope without nested procedures

- A procedure definition cannot occur within another
- Therefore, all non local references are global and can be allocated at compile time
- Any name non-local to one procedure is non-local to all procedures

Lexical scope without nested procedures

- A procedure definition cannot occur within another
- Therefore, all non local references are global and can be allocated at compile time
- Any name non-local to one procedure is non-local to all procedures
- In absence of nested procedures use stack allocation

Lexical scope without nested procedures

- A procedure definition cannot occur within another
- Therefore, all non local references are global and can be allocated at compile time
- Any name non-local to one procedure is non-local to all procedures
- In absence of nested procedures use stack allocation
- Storage for non locals is allocated statically

Lexical scope without nested procedures

- A procedure definition cannot occur within another
- Therefore, all non local references are global and can be allocated at compile time
- Any name non-local to one procedure is non-local to all procedures
- In absence of nested procedures use stack allocation
- Storage for non locals is allocated statically
- Stack allocation of non local has advantage:

Lexical scope without nested procedures

- A procedure definition cannot occur within another
- Therefore, all non local references are global and can be allocated at compile time
- Any name non-local to one procedure is non-local to all procedures
- In absence of nested procedures use stack allocation
- Storage for non locals is allocated statically
- Stack allocation of non local has advantage:
  - Non locals have static allocations

Lexical scope without nested procedures

- A procedure definition cannot occur within another
- Therefore, all non local references are global and can be allocated at compile time
- Any name non-local to one procedure is non-local to all procedures
- In absence of nested procedures use stack allocation
- Storage for non locals is allocated statically
- Stack allocation of non local has advantage:
  - Non locals have static allocations
  - Procedures can be passed/returned as parameters

Scope with nested procedures

- Nesting Depth

Scope with nested procedures

- Nesting Depth
  - Main procedure is at depth 1

Scope with nested procedures

- Nesting Depth
  - Main procedure is at depth 1
  - Add 1 to depth as we go from enclosing to enclosed procedure

Scope with nested procedures

- Nesting Depth
  - Main procedure is at depth 1
  - Add 1 to depth as we go from enclosing to enclosed procedure
- Access to non-local names

Scope with nested procedures

- Nesting Depth
  - Main procedure is at depth 1
  - Add 1 to depth as we go from enclosing to enclosed procedure
- Access to non-local names
  - Include a field `access link` in the activation record

Scope with nested procedures

- Nesting Depth
  - Main procedure is at depth 1
  - Add 1 to depth as we go from enclosing to enclosed procedure
- Access to non-local names
  - Include a field `access link` in the activation record
  - If p is nested in q then access link of p points to the access link in most

Scope with nested procedures

- Nesting Depth
  - Main procedure is at depth 1
  - Add 1 to depth as we go from enclosing to enclosed procedure
- Access to non-local names
  - Include a field `access link` in the activation record
  - If p is nested in q then access link of p points to the access link in most
  - Suppose procedure p at depth n_p refers to a non-local a at depth n_a, then storage for a can be found as

Scope with nested procedures

- Nesting Depth
  - Main procedure is at depth 1
  - Add 1 to depth as we go from enclosing to enclosed procedure
- Access to non-local names
  - Include a field `access link` in the activation record
  - If p is nested in q then access link of p points to the access link in most
  - Suppose procedure p at depth n_p refers to a non-local a at depth n_a, then storage for a can be found as
    - follow (n_p-n_a) access links from the record at the top of the stack

Scope with nested procedures

- Nesting Depth
  - Main procedure is at depth 1
  - Add 1 to depth as we go from enclosing to enclosed procedure
- Access to non-local names
  - Include a field `access link` in the activation record
  - If p is nested in q then access link of p points to the access link in most
  - Suppose procedure p at depth n_p refers to a non-local a at depth n_a, then storage for a can be found as
    - follow (n_p-n_a) access links from the record at the top of the stack
    - after following (n_p-n_a) links we reach procedure for which a is local

Paramter Passing

- Call by value

Paramter Passing

- Call by value
  - actual parameters are evaluated and their rvalues are passed to the called procedure used in Pascal and C

Paramter Passing

- Call by value
  - actual parameters are evaluated and their rvalues are passed to the called procedure used in Pascal and C
  - formal is treated just like a local name

Paramter Passing

- Call by value
  - actual parameters are evaluated and their rvalues are passed to the called procedure used in Pascal and C
  - formal is treated just like a local name
  - caller evaluates the actual parameters and places rvalue in the storage for formals

Paramter Passing

- Call by value
  - actual parameters are evaluated and their rvalues are passed to the called procedure used in Pascal and C
  - formal is treated just like a local name
  - caller evaluates the actual parameters and places rvalue in the storage for formals
  - call has no effect on the activation record of caller

Paramter Passing

- Call by value
  - actual parameters are evaluated and their rvalues are passed to the called procedure used in Pascal and C
  - formal is treated just like a local name
  - caller evaluates the actual parameters and places rvalue in the storage for formals
  - call has no effect on the activation record of caller
- Call by reference (call by address)

Paramter Passing

- Call by value
  - actual parameters are evaluated and their rvalues are passed to the called procedure used in Pascal and C
  - formal is treated just like a local name
  - caller evaluates the actual parameters and places rvalue in the storage for formals
  - call has no effect on the activation record of caller
- Call by reference (call by address)
  - the caller passes a pointer to each location of actual parameters

Paramter Passing

- Call by value
  - actual parameters are evaluated and their rvalues are passed to the called procedure used in Pascal and C
  - formal is treated just like a local name
  - caller evaluates the actual parameters and places rvalue in the storage for formals
  - call has no effect on the activation record of caller
- Call by reference (call by address)
  - the caller passes a pointer to each location of actual parameters
  - if actual parameter is a name then lvalue is passed

Paramter Passing

- Call by value
  - actual parameters are evaluated and their rvalues are passed to the called procedure used in Pascal and C
  - formal is treated just like a local name
  - caller evaluates the actual parameters and places rvalue in the storage for formals
  - call has no effect on the activation record of caller
- Call by reference (call by address)
  - the caller passes a pointer to each location of actual parameters
  - if actual parameter is a name then lvalue is passed
  - if actual parameter is an expression then it is evaluated in a new location and the address of that location is passed

Runtime Storage Management

- Run time allocation and de-allocation of activations occurs as part of procedure call and return sequences

Runtime Storage Management

- Run time allocation and de-allocation of activations occurs as part of procedure call and return sequences
- Assume four kind of statements: call, return, halt and action

Runtime Storage Management

- Run time allocation and de-allocation of activations occurs as part of procedure call and return sequences
- Assume four kind of statements: call, return, halt and action
- A call statement is implemented by a sequence of two instructions

Runtime Storage Management

- Run time allocation and de-allocation of activations occurs as part of procedure call and return sequences
- Assume four kind of statements: call, return, halt and action
- A call statement is implemented by a sequence of two instructions
- A move instruction saves the return address

Runtime Storage Management

- Run time allocation and de-allocation of activations occurs as part of procedure call and return sequences
- Assume four kind of statements: call, return, halt and action
- A call statement is implemented by a sequence of two instructions
- A move instruction saves the return address
- A goto transfers control to the target code

Runtime Storage Management

- Run time allocation and de-allocation of activations occurs as part of procedure call and return sequences
- Assume four kind of statements: call, return, halt and action
- A call statement is implemented by a sequence of two instructions
- A move instruction saves the return address
- A goto transfers control to the target code
- Instruction Sequence can be

Runtime Storage Management

- Run time allocation and de-allocation of activations occurs as part of procedure call and return sequences
- Assume four kind of statements: call, return, halt and action
- A call statement is implemented by a sequence of two instructions
- A move instruction saves the return address
- A goto transfers control to the target code
- Instruction Sequence can be
  - `MOV here+Inst_size callee.static_area`

Runtime Storage Management

- Run time allocation and de-allocation of activations occurs as part of procedure call and return sequences
- Assume four kind of statements: call, return, halt and action
- A call statement is implemented by a sequence of two instructions
- A move instruction saves the return address
- A goto transfers control to the target code
- Instruction Sequence can be
  - `MOV here+Inst_size callee.static_area`
  - `GOTO callee.code_area`

Runtime Storage Management

- Run time allocation and de-allocation of activations occurs as part of procedure call and return sequences
- Assume four kind of statements: call, return, halt and action
- A call statement is implemented by a sequence of two instructions
- A move instruction saves the return address
- A goto transfers control to the target code
- Instruction Sequence can be
  - `MOV here+Inst_size callee.static_area`
  - `GOTO callee.code_area`
  - `callee.static_area` and `callee.code_area` are constants referring to address of the activation record and the first address of called procedure respectively.

Runtime Storage Management

- Run time allocation and de-allocation of activations occurs as part of procedure call and return sequences
- Assume four kind of statements: call, return, halt and action
- A call statement is implemented by a sequence of two instructions
- A move instruction saves the return address
- A goto transfers control to the target code
- Instruction Sequence can be
  - `MOV here+Inst_size callee.static_area`
  - `GOTO callee.code_area`
  - `callee.static_area` and `callee.code_area` are constants referring to address of the activation record and the first address of called procedure respectively.
  - A return from procedure callee is implemented by `GOTO *callee.static_area`

Stack Allocation

- Position of the activation record is not known until run time

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and data are accessed with an offset from SP.

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and data are accessed with an offset from SP.
- The code for the first procedure initializes the stack by setting up SP to the start of the stack area

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and data are accessed with an offset from SP.
- The code for the first procedure initializes the stack by setting up SP to the start of the stack area
  - `MOV #Stackstart, SP`

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and data are accessed with an offset from SP.
- The code for the first procedure initializes the stack by setting up SP to the start of the stack area
  - `MOV #Stackstart, SP`
  - `code for the first procedure`

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and data are accessed with an offset from SP.
- The code for the first procedure initializes the stack by setting up SP to the start of the stack area
  - `MOV #Stackstart, SP`
  - `code for the first procedure`
  - `HALT`

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and data are accessed with an offset from SP.
- The code for the first procedure initializes the stack by setting up SP to the start of the stack area
  - `MOV #Stackstart, SP`
  - `code for the first procedure`
  - `HALT`
- A procedure call sequence increments SP, saves the return address and transfers control to the called procedure

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and data are accessed with an offset from SP.
- The code for the first procedure initializes the stack by setting up SP to the start of the stack area
  - `MOV #Stackstart, SP`
  - `code for the first procedure`
  - `HALT`
- A procedure call sequence increments SP, saves the return address and transfers control to the called procedure
  - `ADD #caller.recordsize, SP`

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and data are accessed with an offset from SP.
- The code for the first procedure initializes the stack by setting up SP to the start of the stack area
  - `MOV #Stackstart, SP`
  - `code for the first procedure`
  - `HALT`
- A procedure call sequence increments SP, saves the return address and transfers control to the called procedure
  - `ADD #caller.recordsize, SP`
  - `MOVE #here+ fixedOffset, *SP`

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and data are accessed with an offset from SP.
- The code for the first procedure initializes the stack by setting up SP to the start of the stack area
  - `MOV #Stackstart, SP`
  - `code for the first procedure`
  - `HALT`
- A procedure call sequence increments SP, saves the return address and transfers control to the called procedure
  - `ADD #caller.recordsize, SP`
  - `MOVE #here+ fixedOffset, *SP`
  - `GOTO callee.code_area`

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and data are accessed with an offset from SP.
- The code for the first procedure initializes the stack by setting up SP to the start of the stack area
  - `MOV #Stackstart, SP`
  - `code for the first procedure`
  - `HALT`
- A procedure call sequence increments SP, saves the return address and transfers control to the called procedure
  - `ADD #caller.recordsize, SP`
  - `MOVE #here+ fixedOffset, *SP`
  - `GOTO callee.code_area`
- return sequence consists of two parts

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and data are accessed with an offset from SP.
- The code for the first procedure initializes the stack by setting up SP to the start of the stack area
  - `MOV #Stackstart, SP`
  - `code for the first procedure`
  - `HALT`
- A procedure call sequence increments SP, saves the return address and transfers control to the called procedure
  - `ADD #caller.recordsize, SP`
  - `MOVE #here+ fixedOffset, *SP`
  - `GOTO callee.code_area`
- return sequence consists of two parts
  - GOTO *0(SP)

Stack Allocation

- Position of the activation record is not known until run time
- Position is stored in a register at run time, and data are accessed with an offset from SP.
- The code for the first procedure initializes the stack by setting up SP to the start of the stack area
  - `MOV #Stackstart, SP`
  - `code for the first procedure`
  - `HALT`
- A procedure call sequence increments SP, saves the return address and transfers control to the called procedure
  - `ADD #caller.recordsize, SP`
  - `MOVE #here+ fixedOffset, *SP`
  - `GOTO callee.code_area`
- return sequence consists of two parts
  - GOTO *0(SP)
  - SUB #caller.recordsize, SP