

The Runtime Environment

- Remember basic architecture of compiler
 - Front end – scanning, parsing, semantic analysis
 - Independent of target machine
 - Back end – optimizing, code generation
 - Depends on target machine
- However, target environments share many characteristics
 - Possible to describe code execution environment independent of a specific target machine
 - This is called the **runtime environment**

Spring 2008

CIS 461/561 - Runtime Environment

1

Memory Management

- The runtime environment is all about use of memory
 - Memory for variables
 - Memory for function arguments
 - Memory for temporaries
- Memory needs depend on language characteristics
 - Scope and lifetime of variables (dynamic vs static scope)
 - Literal values, constants
 - Functions
 - Are they recursive? Can they be nested? Parameters?
 - . . .

Spring 2008

CIS 461/561 - Runtime Environment

2

Kinds of Runtime Environments

- Fully Static
 - Variables have fixed locations, statically allocated
- Stack Based
 - Variables put on processor stack during execution
- Fully Dynamic (Heap)
 - Variables dynamically allocated in memory during program execution under program control

Spring 2008

CIS 461/561 - Runtime Environment

3

Runtime Environment Examples

- Fortran, TINY
 - Fully static – all variables global in TINY
- Scheme
 - Fully Dynamic – all variables are pointers to heap
- Java
 - All objects dynamic, but primitives on stack
- C/C++
 - Uses all three types: global variables, local variables on stack, new'd variables on heap

Spring 2008

CIS 461/561 - Runtime Environment

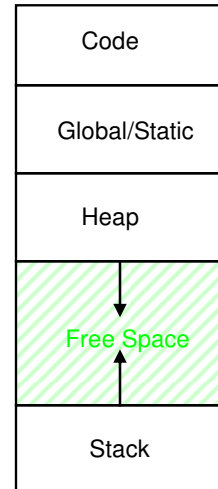
4

Typical Program Layout

```

int x = 1; // global
g() {
  int * x = new int[10]; // heap
}
f(int x) { // stack
  static int x; // static
  {
    int x; // stack
    g();
  }
}
main() {
  int x; // stack
  f(x);
}

```



Spring 2008

CIS 461/561 - Runtime Environment

5

C Minus

- Global variables
 - Use static storage
- Local variables
 - Use stack storage (could we use static?)
- No pointers, no malloc, no new
 - No need for heap

Spring 2008

CIS 461/561 - Runtime Environment

6

Functions

- Independent of language, functions need:
 - Space for parameter values
 - Space for bookkeeping such as return address
 - Space for local variables
 - Space for temporaries
- Kept in an **activation record**
 - Record created for each **call** to function
 - Sometimes called a stack frame

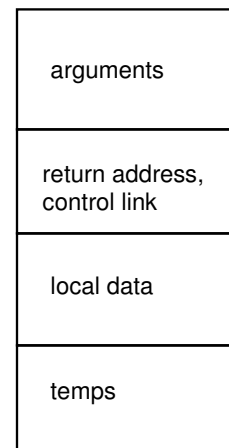
Spring 2008

CIS 461/561 - Runtime Environment

7

Activation Record

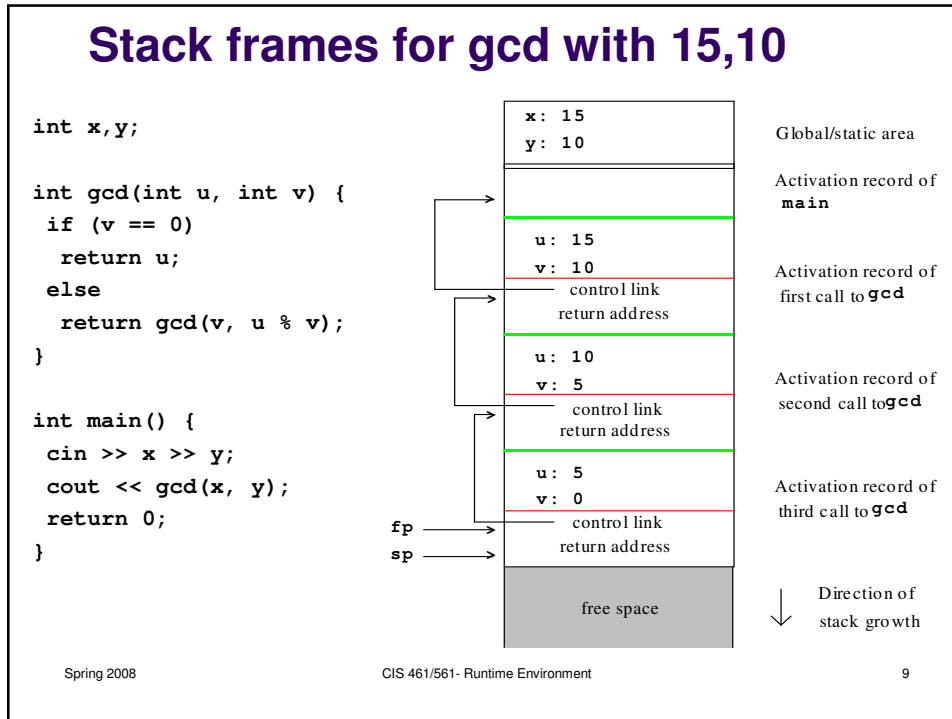
- With each call, push record on stack
- Pop record at return
- Frame pointer (fp) - current record
- Stack pointer (sp) - top of stack
- Program counter (pc) - current instruction



Spring 2008

CIS 461/561 - Runtime Environment

8



Calling Sequence

- Caller
 - Evaluates and fills in parameter values (push on stack)
 - Push return address on stack
 - Jumps to the function
 - On return, copies return value, pops parameters
- Callee
 - Push control link (current fp) on stack – this creates new record
 - Change fp to new record
 - Sees formal parameters in its scope
 - Pushes local variables, temps on stack
 - On exit, pop locals, restore fp from control link
- Note that the parameters could be considered part of old frame as well as new frame
- Frame pointer typically points between parameters and local variables

Calling Sequence

```
void foo() { bar(13, 7); }
void bar(int m, int n) { }
```

```
-----
foo:
    pushl   %ebp           } push fp, make stack new fp
    movl   %esp, %ebp     } grow stack for parameters
    subl   $8, %esp
    movl   $7, 4(%esp)    } copy values to parameters
    movl   $13, (%esp)
    call   _bar           } make the call to bar
    leave
    ret                  } restore stack, restore fp

_bar:
    pushl   %ebp           } push fp
    movl   %esp, %ebp     } make the stack the new fp
    popl   %ebp           } restore it all
    ret
```

Spring 2008

CIS 461/561 - Runtime Environment

11

Accessing Parameters and Locals

For a typical processor managed stack which grows from higher to lower memory addresses...

- Access is by **fixed offset** from frame pointer
 - We know the offsets at compile time, but we don't know the frame pointer at compile time
- Positive offset for parameters
 - They are above the fp
 - They must be pushed by caller in reverse order
- Negative offset for locals, temps
 - They are below the fp
 - Pushed by the code of the function

Spring 2008

CIS 461/561 - Runtime Environment

12

Accessing parameters and locals

```
int f(int x, int y) {
    int z = 42;
    return x + y + z;
}
```

```
-----
_f:
    pushl   %ebp
    movl   %esp, %ebp
    subl   $4, %esp
    movl   $42, -4(%ebp)
    movl   12(%ebp), %eax
    addl   8(%ebp), %eax
    addl   -4(%ebp), %eax
    movl   %ebp, %esp
    popl   %ebp
    ret
```

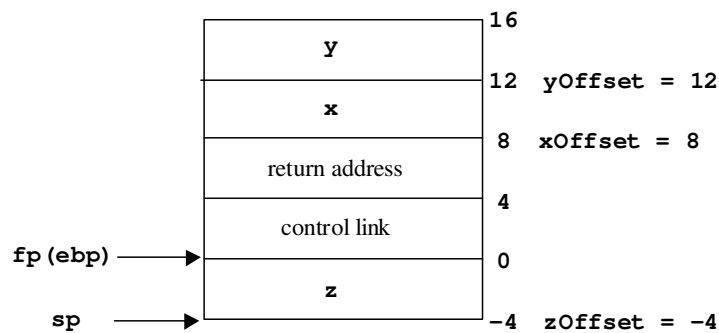
} push fp, make stack new fp
 } "allocate" z on stack
 } init z with 42
 } save y as return
 } add x
 } add z
 } restore stack, restore fp

Spring 2008

CIS 461/561 - Runtime Environment

13

Picture (32-bit architecture):



Spring 2008

CIS 461/561 - Runtime Environment

14

Questions

- Why not allocate entire activation record in caller instead of just the argument part?
 - Caller shouldn't need to know about function's local data
 - Function may grow locals according to execution
 - Provides good separation – caller only needs to know about parameters (which it has in call)
- What about return values?
 - Register used (could treat as a parameter value)
 - Register restricts to scalar types unless pointers are used
- What about return address?
 - Pushed by 'call' instruction, popped by 'ret' instruction
 - Sits between parameters and control link

Spring 2008

CIS 461/561 - Runtime Environment

15

Array allocation on stack

```

int f() {
    int z[4];
    return z[0] + z[2];
}
-----
_f:
    pushl    %ebp
    movl    %esp, %ebp
    subl    $24, %esp           // space for array
    movl    -16(%ebp), %eax     // load z[2]
    addl    -24(%ebp), %eax     // add z[0]
    movl    %ebp, %esp
    popl    %ebp
    ret

```

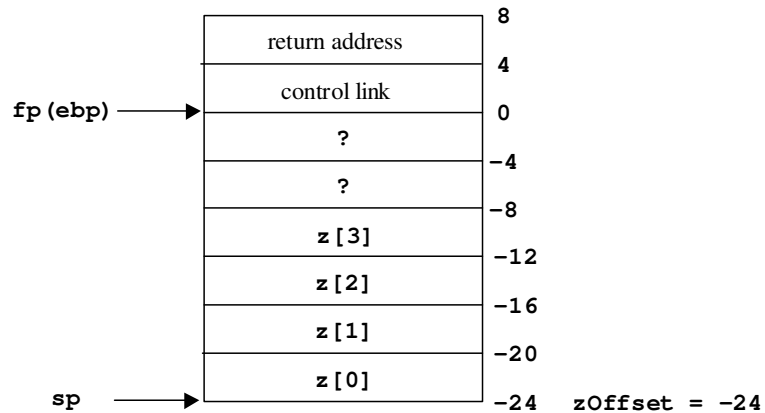
size must be known
at compile time

Spring 2008

CIS 461/561 - Runtime Environment

16

Array Location



Spring 2008

CIS 461/561 - Runtime Environment

17

Computed array subscripts

```
int f(int i) {
    int z[4];
    return z[i];
}
```

```
-----
_f:  pushl %ebp
      movl %esp, %ebp
      subl $24, %esp
      movl 8(%ebp), %eax // move i to eax
      leal 0(,%eax,4), %edx // mult by 4
      leal -24(%ebp), %eax // load z addr
      movl (%edx,%eax), %eax // *(z+4*i) -> eax
      movl %ebp, %esp
      popl %ebp
      ret
```

Spring 2008

CIS 461/561 - Runtime Environment

18

Nested scopes

```

void f() {
  int x;
  {
    int y = 2;
    {
      int z = 3;
      x = y + z;
    }
  }
  {
    int w = 4;
    int v = 5;
    x = v + w;
  }
}

```

```

_f:
  pushl   %ebp
  movl    %esp, %ebp
  subl   $12, %esp
  movl   $2, -8(%ebp)
  movl   $3, -12(%ebp)
  movl   -12(%ebp), %eax
  addl   -8(%ebp), %eax
  movl   %eax, -4(%ebp)
  movl   $4, -12(%ebp)
  movl   $5, -8(%ebp)
  movl   -12(%ebp), %eax
  addl   -8(%ebp), %eax
  movl   %eax, -4(%ebp)
  movl   %ebp, %esp
  popl   %ebp
  ret

```

Spring 2008

CIS 461/561 - Runtime Environment

19

Call by Reference

- Instead of copying values as parameters on stack, copy addresses
 - Callee requires extra level of indirection to get to value
 - But stack is smaller and callee can modify caller's data
- Arrays in C/C++ are passed by reference
 - But arrays in C/C++ are treated as constant pointers
- Objects in Java are passed by reference
 - Well, not quite since assignment in the callee is allowed, but has no effect on the caller's data
- C++ has syntax for true call by reference
 - Probably implemented with pointers, but compiler takes care of making it all work

Spring 2008

CIS 461/561 - Runtime Environment

20

Other Scenarios

- Nested functions
 - Inner function may use local variables of enclosing function
 - Need way to get to local variables of another function
 - Solution: provide **access link** to point to stack frame of enclosing function (set at time of call)
 - To access variables, follow links to correct stack frame
 - May require several link follows (chaining)
- Passing functions
 - Needs closures to capture environment
- Dynamic scope
 - No fixed offset, requires searching at runtime