CS 1622: Activation Records

Jonathan Misurda
jmisurda@cs.pitt.edu

Runtime Considerations

We're moving towards actually producing target code. This means we need to consider the runtime actions of the program.

Runtime support:

- Functions and local variable storage
- Dynamic data allocations
- Garbage collection

For now, we will do this independently of the target language and instead focus on what functions need to work as specified in the source language.

Functions

```
int f(int x) {
    return x;
}

int main() {
    int i;
    for(i = 0; i< 100; i++)
        f(i);
    return 0;
}
```

We need to implement **scope** in terms of the allocation **lifetimes** of our variables.

In languages like C or Java, we have local variables whose lifetime is that of a function call.

However, there are exceptions.

Static Local Variables in C

```
#include <stdio.h>

void f() {
    static int x=0;
    printf("%d\n", ++x);
}

int main() {
    int i;
    for(i = 0; i< 100; i++)
        f(i);
    return 0;
}
```

Static local variables in C are locally-scoped but their allocation lifetime exists longer than the function call.

The solution in C is to treat them like globals.

Higher-order Functions

```
let f x  =
    let g y = x + y
        in g


let h = f(3)
let j = f(4)

let z = h(5)
let w = j(7)

printfn "z is %d w is %d" z w
```

Output:
z is 8 w is 11

This is a higher order function in F# (Microsoft's .NET relative of ML)

The function g is a local **nested function** in f. Nested functions have access to the enclosing function's local variables.

The function f also **returns a function**. This means the function exists longer than the scope and thus its local variables need to have extended lifetimes.

Implementation

Pascal has nested functions, but it does not have functions as returnable values.
C has functions as returnable values, but not nested functions.

Pascal and C can use **stacks** to hold local variables.

F#, ML, Scheme, and several other languages have both nested functions and functions as returnable values so they cannot use stacks to hold all local variables.

Stack

Stack

- A portion of memory managed in a last-in, first-out (LIFO) fashion

Function Call

- A control transfer to a segment of code that ends with a return to the point in code immediately after where the call was made (the return address)

Calling Convention

- An agreement, usually created by a system's designers, on how function calls should be implemented

Activation Records

An object containing all the necessary data for a function

- Values of parameters
- Return address
- Local variables
- Size
- …

Also called a **frame**.

Creation of an activation record occurs in the function **prologue**.

Deletion of an activation record occurs in the function **epilogue**.

This gives the data in the activation record the lifetime of the function's activation.

Why a Stack?

```
int f(int x) {
    return x;
}

int main() {
    int i;
    for(i = 0; i< 100; i++)
        f(i);
    return 0;
}
```

Does this code need a stack?

Alternative: Keep an "array" of activation records.

Does this work?

int i	int x
main	f

Recursion!

```
int f(int x) {
    if(x < 2) return 1;
    return x * f(x-1);
}

int main() {
    int i;
    scanf("%d", &i);
    printf("%d", f(i));
    return 0;
}
```

Now we have an uncertain number of activations that we cannot calculate statically (at compile time).

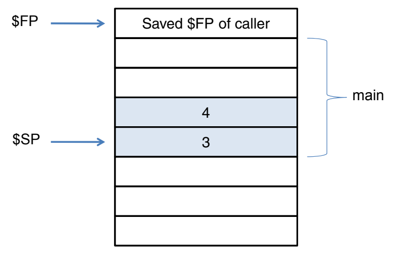
We want as many activation records (and thus copies of x) as necessary to compute our answer to be allocated.

Function Call, 2 Parameters

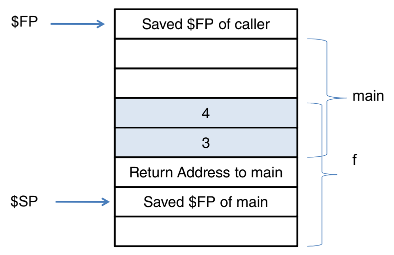
```
#include <stdio.h>

int f(int x, int y)
{
        return x+y;
}

int main()
{
        int y;

        y = f(3, 4);

        return 0;
}
```

```
f:      pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        leave
        ret
main:   pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $4, 4(%esp)
        movl    $3, (%esp)
        call    f
        movl    %eax, 4(%esp)
        movl    $0, %eax
        leave
        ret
```

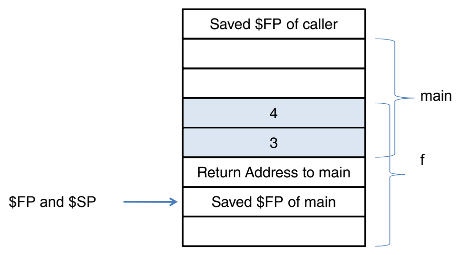
Stack



Stack



Stack



Frame Pointer

We use the **stack pointer** (usually an architectural register) to mark the dividing line between the top of the stack and the free space.

In the epilogue of a method, we need to know how large the AR was in order to pop it off the stack.

We can use a **frame pointer** to mark the bottom of the AR with the stack pointer marking the top.

If we know the size of an activation record at compile time, we can omit the frame pointer and just encode the size directly in the prologue and epilogue.

However, frame pointers can be useful as the size of a frame is generally not known until late during the code generation phase, and so using the frame pointer gives us easy access to the locals and actual parameters.

-fomit-frame-pointer

```
#include <stdio.h>

int f(int x, int y)
{
        return x+y;
}

int main()
{
        int y;

        y = f(3, 4);

        return 0;
}
```

```
f:      movl    8(%esp), %eax
        addl    4(%esp), %eax
        ret

main:   pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $4, 4(%esp)
        movl    $3, (%esp)
        call    f
        movl    $0, %eax
        leave
        ret
```

Registers

Register-register architectures like MIPS require our operands to be loaded into registers before we can perform operations on them.

Register-memory architectures like x86 may allow for a single memory operand to our ALU operations, but operands in registers will still execute faster.

Since we usually only have one set of machine registers, functions must share:

Caller-Saved Registers
- A piece of data (e.g., a register) that must be explicitly saved if it needs to be preserved across a function call

Callee-Saved Registers
- A piece of data (e.g., a register) that must be saved by a called function before it is modified, and restored to its original value before the function returns

MIPS Calling Convention

First 4 arguments $a0-$a3
- Remainder put on stack

Return values $v0-$v1

$t0-$t9 are caller-saved temporaries
$s0-$s9 are callee-saved

C Calling Convention

All parameters must be contiguously laid out in memory (none in registers).

Arguments in memory supports taking the address-of any parameter

- We cannot take the address of a register.

Contiguous layout supports the variadic functions like printf:

- We can walk to the variables on the stack to find our additional arguments at runtime.

Arguments are pushed right to left onto the stack.

Return Addresses

In MIPS, our `JAL` instruction saves the return address (`$PC+4`) into an architectural register, $RA.

In x86, the `CALL` instruction pushes the return address directly onto the stack.

Which one is better?

It seems that x86 saves us a step since we often need to push $RA onto the stack.

However, this is not always the case. If we consider the **call graph** of all possible function calls that may be currently active, we find many functions may be **leaf procedures**.

Leaf procedures do not need to save $RA onto the stack, thus avoiding some memory accesses at runtime. MIPS gives the flexibility of choosing when to spill.

Implementing ARs in MiniJava

```
void f(int x, int y, int z) {
    int a;
    int b;
    int c;
    …
}
```

None of our parameters **escape** the function: They are not passed by reference, have their address taken, or referenced in a nested function. This means that they can be located anywhere in registers or memory.

We will leave the task of where things are allocated to a later phase: the **register allocator**.

MiniJava ARs

	MIPS	x86
x	Register (T_χ)	FP + 8
y	Register (T_ψ)	FP + 12
z	Register (T_ω)	FP + 16
a	Register (T_α)	FP - 4
b	Register (T_β)	FP - 8
c	Register (T_γ)	FP - 12

Temporaries

Some quantities need to be temporarily stored in registers.

The register allocator will decide how to map those temporary values to registers.

For now, we may assume that we have an infinite register set.

The register allocator may have to **spill** values to the stack to accommodate the temporaries that we need.

View Shifts

MIPS	x86

`sub $sp, $sp, AR_SIZE`	`pushl %ebp`
	`movl %esp, %ebp`
	`subl $AR_SIZE, %esp`