Chapter 10
Memory Model for Program Execution

Original slides by Chris Wilcox, Colorado State University
Problem

How do we allocate memory during the execution of a program written in C?

- Programs need memory for code and data such as instructions, global and local variables, etc.
- Modern programming practices encourage many (reusable) functions, callable from anywhere.
- Some memory can be statically allocated, since the size and type is known at compile time.
- Some memory must be allocated dynamically, size and type is unknown at compile time.
Motivation

Why is memory allocation important? Why not just use a memory manager?

- Allocation affects the performance and memory usage of every C, C++, Java program.
- Current systems do not have enough registers to store everything that is required.
- Memory management is too slow and cumbersome to solve the problem.
- Static allocation of memory resources is too inflexible and inefficient, as we will see.
Goals

What do we care about?

- Fast program execution
- Efficient memory usage
- Avoid memory fragmentation
- Maintain data locality
- Allow recursive calls
- Support parallel execution
- Minimize resource allocation
- Memory should never be allocated for functions that are not executed.
Function Call

Consider the following code:

```c
// main program
int a = 10;
int b = 20
int c = foo(a, b);
int foo(int x, int y)
{
    int z;
    z = x + y;
    return z;
}
```

What needs to be stored?
- Code, parameters, locals, globals, return values
Storage Requirements

- Code must be stored in memory so that we can execute the function.
- The return address must be stored so that control can be returned to the caller.
- Parameters must be sent from the caller to the callee so that the function receives them.
- Return values must be sent from the callee to the caller, that’s how results are returned.
- Local variables for the function must be stored somewhere, is one copy enough?
Possible Solution:
Mixed Code and Data

Function implementation:

```assembly
foo        BR foo_begin    # skip over data
foo_rv     .BLKW 1        # return value
foo_ra     .BLKW 1        # return address
foo_paramx .BLKW 1        # 'x' parameter
foo_paramy .BLKW 1        # 'y' parameter
foo_localz .BLKW 1        # 'z' local
foo_begin  ST R7, foo_ra  # save return
            ...
            LD R7, foo_ra  # restore return
            RET
```

Can construct data section by appending foo_
Possible Solution: Mixed Code and Data

Calling sequence

- ST R1, foo_paramx # R1 has ‘x’
- ST R2, foo_paramy # R2 has ‘y’
- JSR foo # Function call
- LD R3, foo_rv # R3 = return value

Code generation is relatively simple.

Few instructions are spent moving data.
Possible Solution: Mixed Code and Data

Advantages:
- Code and data are close together
- Conceptually easy to understand
- Minimizes register usage for variables
- Data persists through life of program

Disadvantages:
- Cannot handle recursion or parallel execution
- Code is vulnerable to self-modification
- Consumes resource for inactive functions
Possible Solution: Separate Code and Data

Memory allocation:

```
foo_rv .BLKW 1  # foo return value
foo_ra .BLKW 1  # foo return address
foo_paramx .BLKW 1  # foo ‘x’ parameter
foo_paramy .BLKW 1  # foo ‘y’ parameter
foo_localz .BLKW 1  # foo ‘z’ local
bar_rv .BLKW 1  # bar return value
bar_ra .BLKW 1  # bar return address
bar_paramw .BLKW 1  # bar ‘w’ parameter
```

- Code for foo() and bar() are somewhere else
- Function code call is similar to mixed solution
Possible Solution: Separate Code and Data

Advantages:
- Code can be marked ‘read only’
- Conceptually easy to understand
- Early Fortran used this scheme
- Data persists through life of program

Disadvantages:
- Cannot handle recursion or parallel execution
- Consumes resource for inactive functions
Real Solution: Execution Stack

- Instructions are stored in code segment
- Global data is stored in data segment
- Statically allocated memory uses stack
- Dynamically allocated memory uses heap

<table>
<thead>
<tr>
<th>Code</th>
<th>Code segment is write protected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Initialized and uninitialized globals</td>
</tr>
<tr>
<td>Heap</td>
<td>Heap can be fragmented</td>
</tr>
<tr>
<td>Stack</td>
<td>Stack size is usually limited</td>
</tr>
<tr>
<td></td>
<td>Stack can grow either direction (usual convention is down)</td>
</tr>
</tbody>
</table>
Execution Stack

What is a stack?

- First In, Last Out (FILO) data structure
- PUSH adds data, POP removes data
- Overflow condition: push when stack full
- Underflow condition: pop when stack empty
- Stack grows and shrinks as data is added and removed
- Stack grows downward from the end of memory space
- Function calls allocate a stack frame
- Return cleans up by freeing the stack frame
- Corresponds nicely to nested function calls
- Stack Trace shows current execution (Java/Eclipse)
Stack Trace

Example stack trace from gdb: main() calls A() calls B() calls C() calls D().

Breakpoint is set in function D(), note that main() is at the bottom, D() is at the top.

(gdb) info stack
#0  D (a=8, b=9) at stacktest.c:23
#1  0x00400531 in C (a=7, b=8) at stacktest.c:19
#2  0x0040050c in B (a=6, b=7) at stacktest.c:15
#3  0x004004e7 in A (a=5, b=6) at stacktest.c:11
#4  0x00400566 in main () at stacktest.c:29
Execution Stack

Picture of stack during program execution, same call stack as previous slide:
- main() calls A(5,6)
- A(5,6) calls B(6,7)
- B(6,7) calls C(7,8)
- C(7,8) calls D(8,9)
Stack Requirements

Consider what has to happen in a function call:

- **Caller** passes arguments to **Callee**
- **Caller** invokes subroutine (JSR).
- **Callee** allocates space for return value.
- **Callee** executes function code.
- **Callee** stores result into return value space.
- **Callee** returns (JMP R7).
- **Caller** loads return value.

Parameters, return value, return address, and locals are stored on the stack.

The order above determines the responsibility and order of stack operations.
**Execution Stack**

**Definition:** A stack frame or activation record is the memory required for a function call:

- Stack frame below contains the function that called this function.
- Stack frame above contains the functions called from this function.
- Caller pushes parameters.
- Callee allocates the return value, saves the return address, allocates/frees local variables, and stores the return value.

| ↑ | Locals |
|   | Return Address |
|   | Return Value |
|   | Parameters |
| ↓ |
Stack Pointers

Clearly we need a variable to store the stack pointer (SP), LC3 assembly uses R6.

Stack execution is ubiquitous, so hardware has a stack pointer, sometimes even instructions.

Problem: stack pointer is difficult to use to access data, since it moves around constantly.

Solution: allocate another variable called a frame pointer (FP), for stack frame, uses R5.

Where should frame pointer point? Our convention sets it to point to the first local variable.
Execution Stack

Definition: A stack frame or activation record is the memory required for a function call:

- Locals are accessed by negative offsets from frame pointer.
- Parameters and return value are accessed by positive offsets.
- Most offsets are small, this explains LDR/STR implementation.
- Base register stores pointer, signed offset accesses both directions.
Execution Stack

- In the previous solutions, the compiler allocated parameters and locals in fixed memory locations.
- Using an execution stack means parameters and locals are constantly moving around.
- The frame pointer solves this problem by using fixed offsets instead of addresses.
- The compiler can generate code using offsets, without knowing where the stack frame will reside.
- Frame pointer needs to be saved and restored around function calls. How about the stack pointer?
Nested Calls

Definition: A stack frame or activation record is the memory required for a function call:

- Locals are accessed by negative offsets from frame pointer.
- Parameters and return value are accessed by positive offsets.
- Most offsets are small, this explains LDR/STR implementation.
- Base register stores pointer, signed offset accesses both directions.
Execution Stack

Advantages:
- Code can be marked ‘read only’
- Conceptually easy to understand
- Supports recursion and parallel execution
- No resources for inactive functions
- Good data locality, no fragmenting
- Minimizes register usage

Disadvantages:
- More memory than static allocation
Detailed Example

Assume POP and PUSH code as follows:

MACRO PUSH(reg)
    ADD R6, R6, #-1 ; Decrement SP
    STR reg, R6, #0 ; Store value
END

MACRO POP(reg)
    LDR reg, R6, #0 ; Load value
    ADD R6, R6, #1 ; Increment SP
END
Detailed Example

Main program to illustrate stack convention:

```assembly
.ORIG x3000

MAIN    LD R6,STACK    ; init stack pointer
         LD R0,OPERAND0 ; load first operand
         PUSH R0        ; PUSH first operand
         LD R1,OPERAND1 ; load second operand
         PUSH R1        ; PUSH second operand
         JSR FUNCTION   ; call function
         LDR R0,R6,#0   ; POP return value
         ADD R6,R6,#3   ; unwind stack
         ST  R0,RESULT  ; store result
         HALT
```


Detailed Example

Stack before JSR instruction
Detailed Example

Function code to illustrate stack convention:

FUNCTION

ADD R6,R6,#-1 ; alloc return value
PUSH R7 ; PUSH return address
PUSH R5 ; PUSH frame pointer
ADD R5,R6,#-1 ; FP = SP-1

ADD R6,R6,#-1 ; alloc local variable
LDR R2,R5,#4 ; load first operand
LDR R3,R5,#5 ; load second operand
ADD R4,R3,R2 ; add operands
STR R4,R5,#0 ; store local variable
Detailed Example

Stack during body of FUNCTION

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Variable</td>
<td>Frame Pointer</td>
<td>Return Address</td>
<td>Return Value</td>
<td>Second Operand</td>
<td>First Operand</td>
</tr>
</tbody>
</table>

FP
Detailed Example

Function code to illustrate stack convention:

```assembly
FUNCTION ; stack exit code
    STR R4,R5,#3 ; store return value
    ADD R6,R5,#1 ; SP = FP+1
    POP R5      ; POP frame pointer
    POP R7      ; POP return address
    RET         ; return

OPERAND0   .FILL x1234 ; first operand
OPERAND1   .FILL x2345 ; second operand
RESULT     .BLKW 1      ; result
STACK       .FILL x4000 ; stack address
```
Stack Execution

Summary of memory model:
- We have discussed the stack model for execution of C programs, and along the way we have shown how a compiler might generate code for function calls.

Future programming assignment:
- Write a recursive function in C, then implement the same function in assembly code, managing memory using the stack model.