COMPUTER SYSTEMS ORGANIZATION

Subroutines in ARM -- Spring 2010 -- IIIT-H -- Suresh Purini

Subroutines, Procedures and Functions

Subroutines, Procedures and Functions – Are they any different?

Why Procedures?

- Avoid duplication of code within the same program.
- Reuse of code across different programs (Libraries?)
- Decomposition of a Complex Program into a set of more manageable subroutines.
- Information Hiding.
- ….
- Any Disadvantages?

Implementing Subroutine Abstraction in ARM

- Question: Can we implement subroutines in ARM using the instructions we have seen so far (ignoring the BL instruction)?

```
main()
{
   int i, sum = 0;
   for(i =0; i<100; ++i )
             sum = sum +sqr(i);
}

int sqr(int n )
{
  return n*n;
}
```

Questions:

1. How to implement Control Transfer?
2. How to pass Parameters?
3. How to pass back Return Values?

Subroutines – Implementing Control Transfer

```
main:
          .
          .
          .
          b sum
label:
          .
          .
          .
sum:
          .
          .
          .
          b label
```

Questions:

1. What is the problem with this approach?
2. How can it be resolved?

Subroutines – Implementing Control Transfer

```
main:
        .
        .
        .
        bl sum      ; branch to sum
        .           ; return here
        .
        .
sum:
        .
        .
        .
        mov pc, lr   ; return
```

Return address will be stored in the Link Register r14 (lr).

B and BL Instruction Format

31–28	27–25	24	23–0
cond	1 0 1	L	24-bit signed word offset

Subroutines – Implementing Control Transfer

```
main() {
   int i, sum = 0;
   for(i =0; i<100; ++i )
            sum = sum +sqr(i);
}

int sqr(int n ) {
  return mult(n, n);
}

int mult(int a, int b){
  return a*b;
}
```

```
main:
          .
          .
          .
          bl sum      ; branch to sum
                      ; return here
          .
          .
          .
sum:
          .
          bl mult     ; branch to mult
          .
          .
          mov pc, lr  ; return
mult:
          .
          .
          mov pc, lr  ; return
```

Does this program work ?

Stacks and Subroutines – Implementing Control Transfer

```
main:
        .
        .
        sub sp, sp, #4
        str lr, [sp]
        bl sum
        ldr lr, [sp]
        add sp, sp, #4
        .
        .
sum:    .
        .
        sub sp, sp, #4
        str lr, [sp]
        bl mult    ; branch to mult
        ldr lr, [sp]
        add sp, sp, #4
        .
        .
        mov pc, lr    ; return
mult:
        .
        .
        mov pc, lr    ; return
```

Subroutine Calling Sequence:

1. Store the Link Register on the stack and adjust the stack Pointer
2. Call the Procedure
3. Pop the contents of the Link Register back from the stack and adjust the stack pointer.

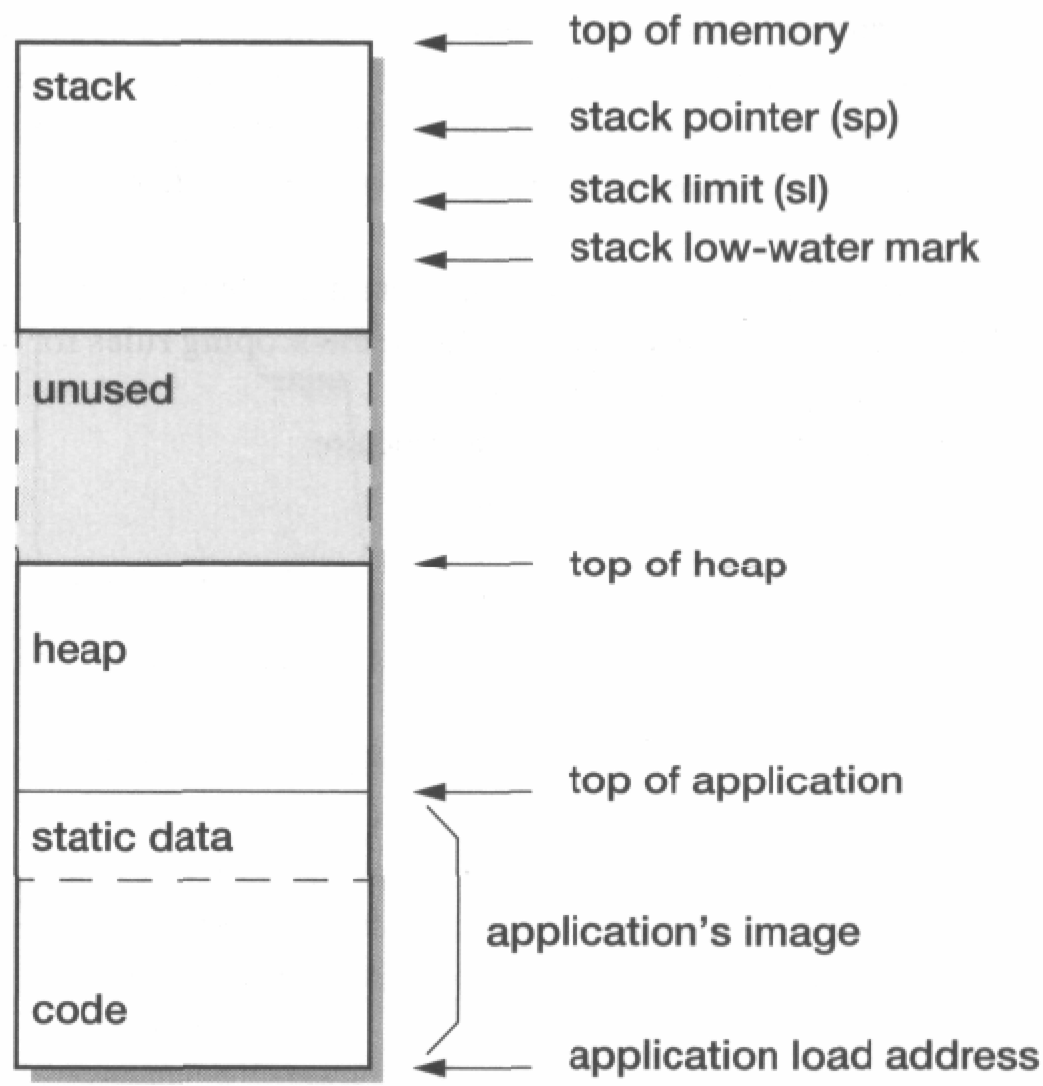
Caller

Agreement: Callee promises Caller to preserve the lr, sp and also the stack contents.

Callee

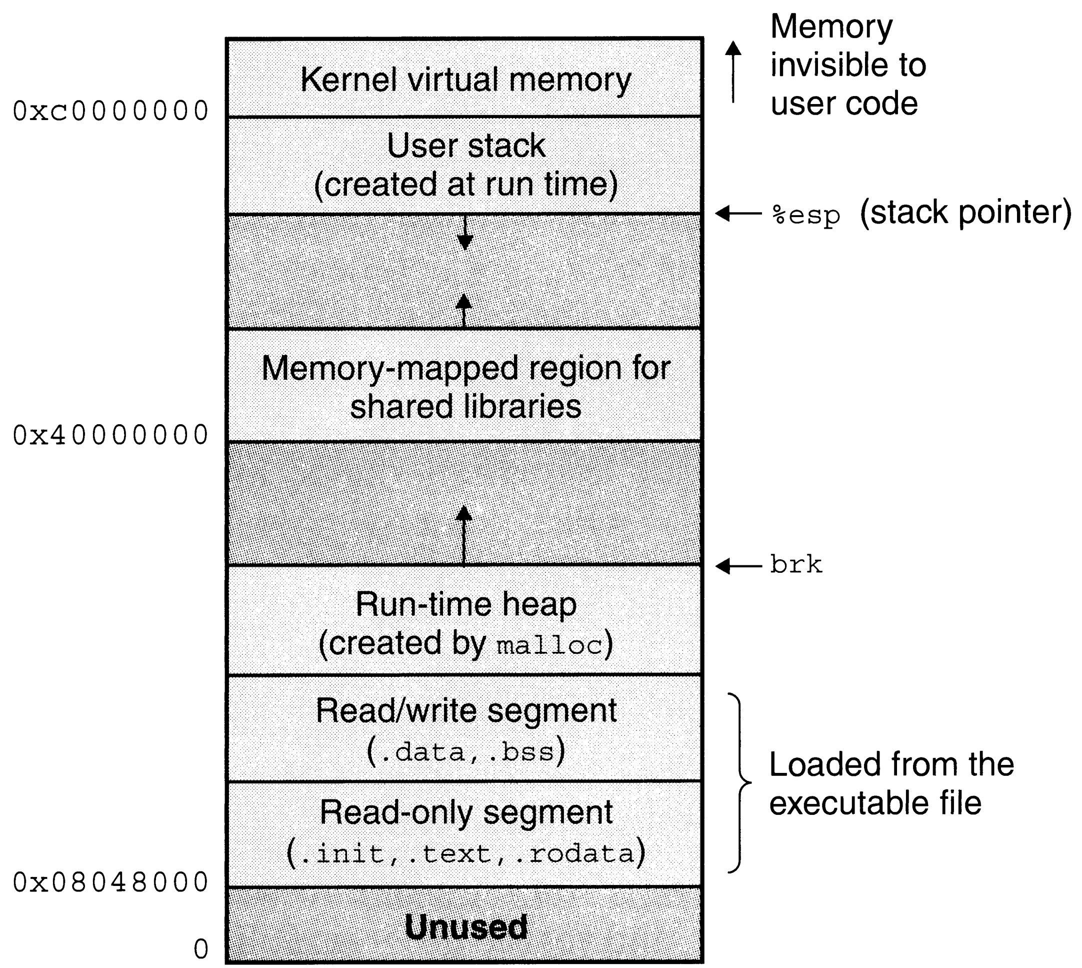
Subroutines – Implementing Control Transfer

Hey, what's happening? Why are we decrementing the sp register when we want to push a return address onto the stack?



Key Point: Stack is growing from higher memory address to lower memory address.
It is a convention.

Linux Memory Image



Subroutines – Implementing Control Transfer

```
main(){
   int n = getchar();
   computerSqrSum(&n);
}

int computerSqrSum(int *pn) {
   int i, sum = 0;
   for(i =0; i<*pn; ++i )
               sum = sum +sqr(i);
   *pn = sum;
    return sum;
}

int sqr(int n ) {
  return n*n;
}
```

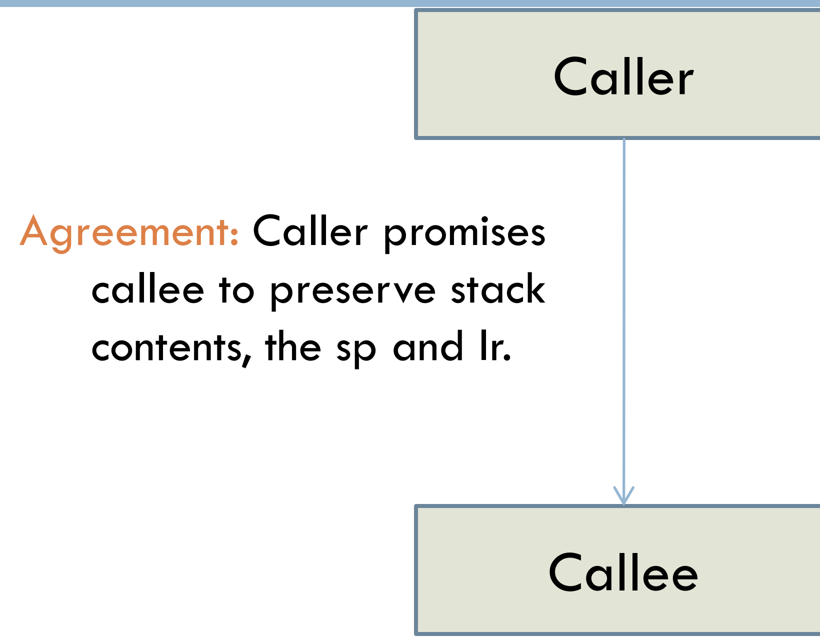
Subroutines – Implementing Control Transfer

```
void printReverse(char *str) {
   char *lstr;
   if( *str) { return; }
   else {
      lstr = str + 1;
      printReverse(lstr)
   }
   putchar(*str);
}
```

Stacks and Subroutines – Implementing Control Transfer

Steps for calling a SubRoutine

1. Store the Link Register on the stack and adjust the stack Pointer.
2. Call the SubRoutine.
3. Pop the contents of the Link Register back from the stack and adjust the stack pointer.



Instead of doing this for every Subroutine Call, why can't we store the link register at the beginning of the Caller code and Restore it at the end.

Passing Parameters and Return Values

How to pass Parameters and Return Values?

1. Pass parameters through registers. If there are more parameters than available registers, pass the rest of the parameters through the stack.
2. Return Values?

Who should preserve the Registers?

- Subroutine foo wants to call Subroutine fun.
- Strategy 1: Before calling fun, foo stores all the registers it would like to preserve on the stack and restore them after the return from the function fun.
- Strategy 2: fun simply calls foo and foo saves all the registers it would like to use on the stack and restores them back before the return to fun.

What are the pros and cons of the two approaches?

Here comes the idea of caller saved registers and callee saved registers.

ARM Procedure Call Standard

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

ARM Procedure Call Standard

Preserved	Not Preserved
Variable Registers: r4 – r11	Argument Registers: r0 – r3
Stack Pointer: sp	Intra-procedure-Call Scratch Register: r12
Link Register: lr	Stack below the stack pointer
Stack above the stack pointer	

1. You may not strictly care for these conventions within your assembly code.
2. However you should pay attention to these guidelines when making library function calls.

Procedure Prologue and Epilogue Code

```
foo:
        sub sp, sp, 16
        str lr, [sp]
        str r4, [sp, #4]
        str r5, [sp, #8]
        str r6, [sp, #12]
        .
        .
        .
        ldr r6, [sp, #12]
        str r5, [sp, #8]
        str r4, [sp, #4]
        str lr, [sp]
        add sp, sp, 16
```

Consider a subroutine foo which not only needs the registers r0-r3, r12, but also requires three additional registers r4, r5, r6 to do its computation.

Preserved	Not Preserved
Variable Registers: r4 – r11	Argument Registers: r0 – r3
Stack Pointer: sp	Intra-procedure-Call Scratch Register: r12
Link Register: lr	Stack below the stack pointer
Stack above the stack pointer	

Procedure Prologue and Epilogue Code

```
foo:
        stmfd sp!, {r4-r6, lr}
        .
        .
        .
        ldmfd sp!, {r4-r6, pc}
```

Did you see this block data transfer instruction?

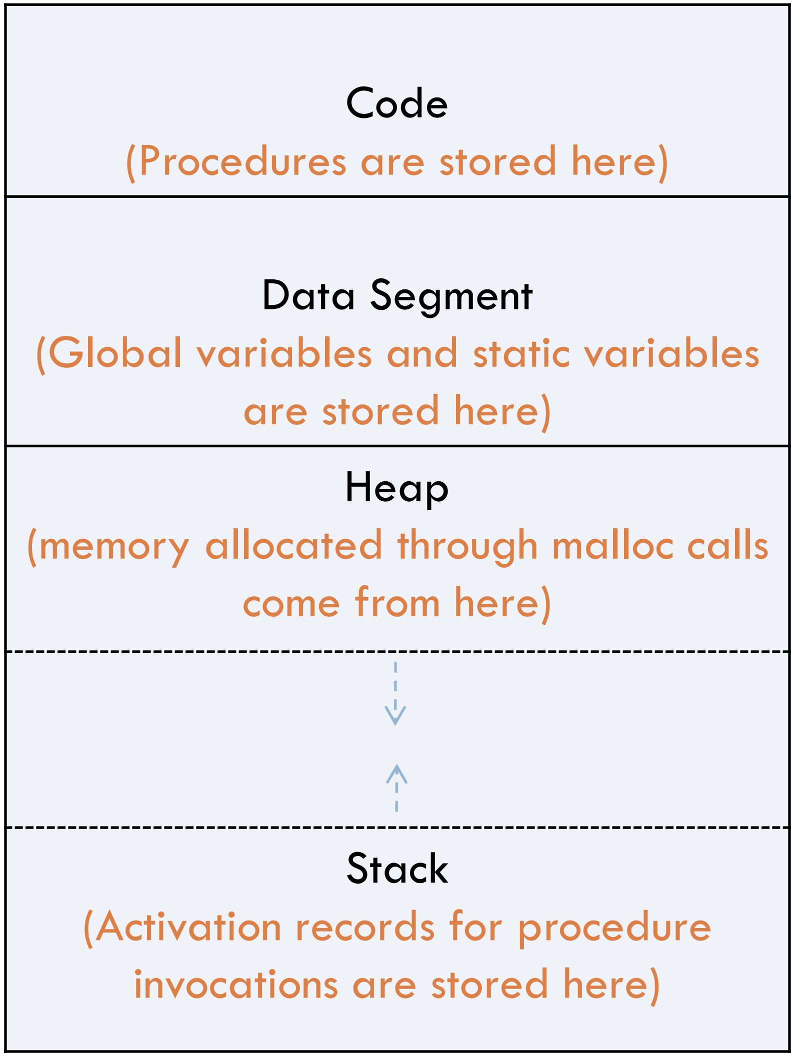
Preserved	Not Preserved
Variable Registers: r4 – r11	Argument Registers: r0 – r3
Stack Pointer: sp	Intra-procedure-Call Scratch Register: r12
Link Register: lr	Stack below the stack pointer
Stack above the stack pointer	

Procedure Calling Sequences

- Procedure main wants to call foo. It passes its parameters through the registers r0 and r1. It is also using registers r2 and r3 for its local computation. So it has to make to sure the register contents r2 and r3 are preserved across the function call to foo. But how?

Preserved	Not Preserved
Variable Registers: r4 – r11	Argument Registers: r0 – r3
Stack Pointer: sp	Intra-procedure-Call Scratch Register: r12
Link Register: lr	Stack below the stack pointer
Stack above the stack pointer	

Storage Lay-out of a Program



Activation Records

- A program is a life-less entity at compile time and in general until it is invoked. It gains life when it is executed.
- Similarly a procedure in a program comes to life when it is invoked or activated.
- Each Live (or Active!) Procedure during a program execution has an associated Data-Structure called Activation Record.
- What does a Live Procedure need for it to carry out it business?

Activation Records

- What does a Live Procedure need for it to carry out it business?
  - Space for its parameters
  - Space for the return value
  - Space for the Local Variables
  - Return address of the Calling Procedure.
  - Space to save the calling procedure context like registers etc.

Actual Parameters
Return Value
Saved Machine Status
Local Variables
Temporary Variables

Actual parameters and return values can be communicated through processor registers also.