Run-time Environment for a Program

different logical parts of a program during execution

- **stack** – automatically allocated variables (local variables, subdivided into stack frames - one per procedure invocation)

- **heap** – dynamically allocated variables

- initialized data (**data**) – global and static variables that are initialized by the programmer. includes **rodata** – read only data

- uninitialized data (**bss**, acronym for **Block Started by Symbol**) – global and static variables that are initialized to zero or do not have explicit initialization in source code.

- **text** - program code / instructions
Run-time Environment for a Program

```
int g = 1; @ g in data
g in data

int main()
{
    int i; @ i alloc. on stack
    ...
}
```

Variables declared once per program

Dynamically allocated variable, e.g., malloc(); new; C pointers

Space for saved procedure information

(note: some memory diagrams may be reversed, with high memory at top)
Memory Stack

• stack matches the last in first out (LIFO) behavior of nested procedure calls (including recursion)

• storage for automatic variables, that is, for local variables that are allocated at the beginning of a subroutine call and that are discarded at the end of the call

• The collection of information related to a specific instance of calling a subroutine is termed a "stack frame"
  – a generic stack frame consists of 1) parameters, 2) return address, 3) registers to save, 4) local variables, and 5) old frame pointer (fp)

• Stack pointer (sp)
  – points to top of stack
Run-time Environment for a Program

- Stack is located in high addresses in memory and grows toward small addresses, so creating space for local variables decrements the stack pointer.
- Subroutines must make room for local variables.
Memory Stack

ARM has the following load and store instructions:

- **ldr** – load word
- **ldrh** – load halfword
- **ldrb** – load byte
- **ldm** – load multiple words
- **str** – store word
- **strh** – store halfword
- **strb** – store byte
- **stm** – store multiple words
Memory Stack

typical form of load for accessing a local variable:

    ldr    rn, [sp, #4]


typical form of store for accessing a local variable:

    str    rs, [sp, #4]
Memory Stack

Loads and stores (example)

```c
/* int main(){
    int a = 0;
    a = a + 1;
}
*/
```

```
a .req r4
add sp, sp, #-4  @ space for local variable
    @ a's address is sp
mov a, #0        @ a = 0; next three instructions
str a, [sp]      @ perform a = a + 1;
ldr a, [sp]      @  1. read a's value (from the stack)
add a, a, #1     @  2. add one
str a, [sp]      @  3. store new value back into a
    @       (into the stack)
mov r0, #0       @ restore the stack pointer
add sp, sp, #4   @ request r4
pop {lr}
bx lr
```
Memory Stack

Pointers

- A pointer is the address of an object in memory.
- A pointer is 64 bits in length (on our current machines).
- A pointer can be in a register or in a memory word.

Example

```c
int main()
{
    int a;
    int *pa = &a;
    *pa = 5;
    return 0;
}
```
int main() {
    int a;
    int *pa = &a;
    *pa = 5;
    return 0;
}

.global main
.type main, %function
main:
push    {r5}
add     sp, sp, #(-12) ; space for local vars
add     r5, sp, #0  ; @ &a in r5
mov     r4, r5      ; @ &a in r4
str     r4, [r5, #4] ; @ &a in pa ([sp+4])
mov     r3, #5      ; @ 5 in r3
str     r3 [r4, #0] ; @ 5 in a ([sp])
mov     r4, #0
mov     r0, r4
add     r5, r5, #12
mov     sp, r5      ; @ restore stack pointer
pop     {r5}
bx      lr
Memory Stack

Extended example with printf stmts

```c
int main(){
    int a = 0;
    int *pa = &a;

    printf("address of a = 0x%08x and contents of a = 0x%08x\n", &a,a);
    printf("address of pa = 0x%08x and contents of pa = 0x%08x\n", &pa,pa);
    *pa = 5;
    printf("address of a = 0x%08x and contents of a = 0x%08x\n", &a,a);
    printf("address of pa = 0x%08x and contents of pa = 0x%08x\n", &pa,pa);
    return 0;
}
```

[23:18:44] rlowe@dragon7:~/cs231-su13/subs [130] ./a.out
address of a = 0xf6fffac8 and contents of a = 0x00000000
address of pa = 0xf6fffacc and contents of pa = 0xf6fffac8
address of a = 0xf6fffac8 and contents of a = 0x00000005
address of pa = 0xf6fffacc and contents of pa = 0xf6fffac8