Lec 08 Run-time Stack

1. 函数调用基础（Basics of Function in C）

- 示例代码

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int rand_gen(int a, int b);
int main()
{
  int a,b,i;
  printf("Enter the rand a and b: ");
  scanf("%d%d",&a, &b);
  srand(time(0));
  for(i=0; i<5; i++){
    printf("%d ", rand_gen(a,b));
    printf("\n");
  }
  return 0;
}
```

其中：

```c
int rand_gen(int a, int b){
  return (rand()%(b-a+1)+a);
}
```

  - 展示了一个简单的C程序，用于生成指定范围内的5个随机数。
  - 代码中使用了 rand_gen 函数，调用时通过参数传递范围 [a, b]，并利用`srand(time(0))`初始化随机种子。
  - 这部分是引入函数调用的基本概念。

- 知识点：
  - 函数通过参数传递数据。
  - 主函数 main 调用子函数 rand_gen，体现了调用者和被调用者（caller 和 callee）的关系。

2. 函数调用的四个基本阶段

- 内容：
  1. 调用者将参数值传递给被调用函数。
  2. 控制权从调用者转移到被调用者。

3. 被调用者执行其任务。
4. 控制权返回调用者，并带回返回值。

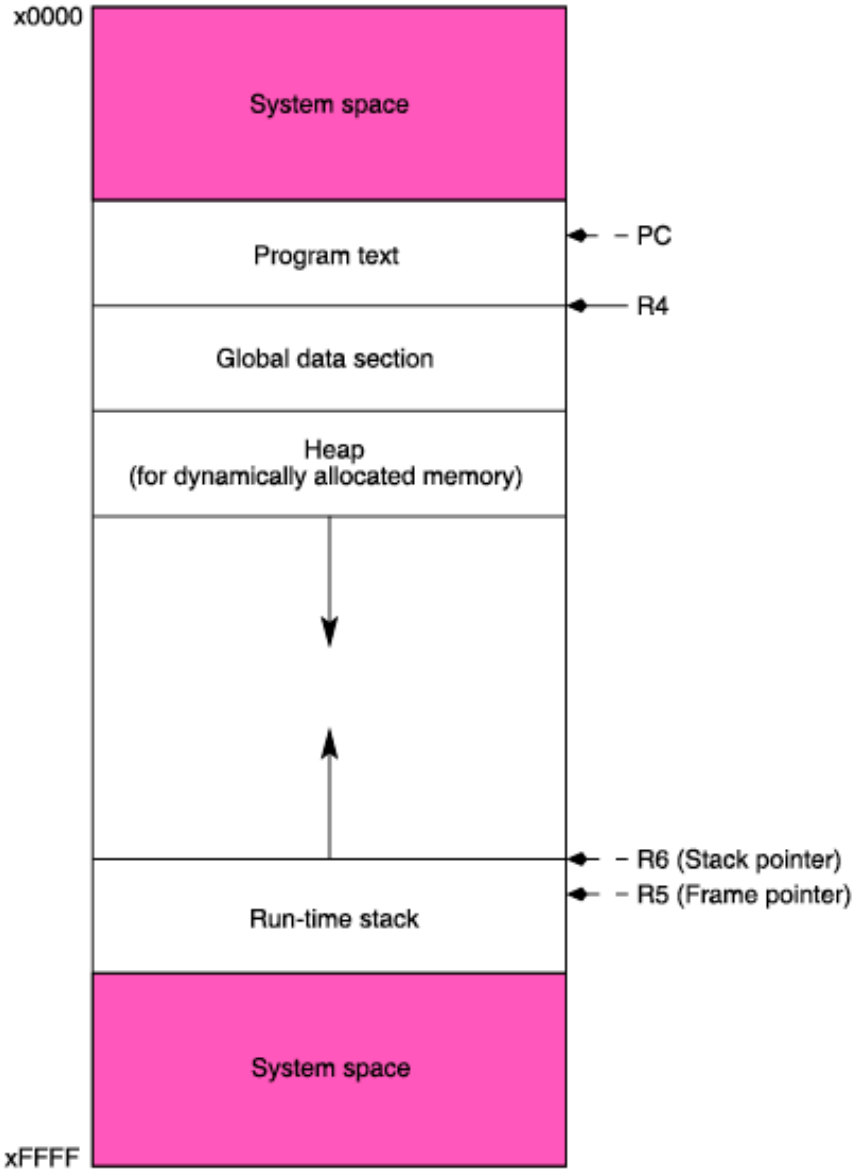
- 知识点：
  - 函数调用是一个有序的过程，涉及参数传递、控制转移和结果返回。
  - 这些阶段需要内存管理支持，而运行时栈正是实现这一机制的关键。

3. 运行时栈与激活记录（Activation Record）

- 内容：
  - 每次函数调用都会创建一个 **激活记录（Activation Record 或 Stack Frame）**，并将其压入运行时栈。
  - 激活记录包含局部变量等信息。
  - 函数返回时，激活记录从栈中弹出。
  - 函数嵌套调用（包括递归）时，栈会增长。
- 知识点：
  - 运行时栈是一个动态的、后进先出（LIFO）的结构。
  - 激活记录是函数执行的“上下文”，保存了函数运行所需的所有信息。
  - 递归调用之所以可行，是因为每次调用都有独立的激活记录。

4. 变量存储空间

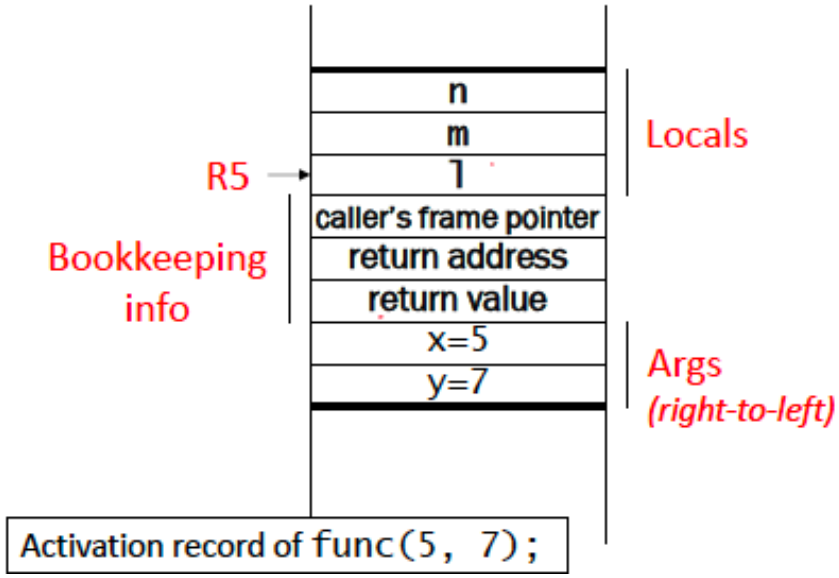
- 内容：
  - 全局变量存储在 **全局数据段（Global Data Section）**。
  - 局部变量存储在 **运行时栈**。
  - 使用寄存器管理栈：
    - **R4（全局指针）**：指向第一个全局变量。
    - **R5（帧指针）**：指向第一个局部变量（运行时栈）。
    - **R6（栈指针）**：指向运行时栈的栈顶。
- 知识点：
  - 内存分为不同的区域，运行时栈专门用于动态分配局部变量。
  - 帧指针和栈指针协同工作，管理栈的增长和缩减。



5. 激活记录的构建

- 内容：
  - 激活记录包括：
    1. 参数（Arguments）。
    2. 书签信息（Bookkeeping Info，如返回地址）。
    3. 局部变量（Local Variables）。
  - 示例代码展示了 main 调用 func，并分配局部变量。

```
int main{
  int x,y,z;
  x = func(5,7);
}
int func(int x, int y){
  int l,m,n;
  // some operations...
  return l;
}
```



- 知识点：
  - 激活记录是一个结构化的内存块，包含函数运行的所有必要信息。
  - 栈的动态分配确保了每次调用都有独立的空间。

6. 栈的构建与拆除过程

- 内容：
  - 函数调用分为7个步骤：
    1. **调用者准备**：将参数压栈。
    2. **控制转移**：跳转到被调用函数（JSR/JSRR）。
    3. **被调用者准备**：压入书签信息和局部变量。
    4. **执行函数**。
    5. **被调用者拆除**：更新返回值，弹出局部变量等。
    6. **返回调用者**：通过 RET 指令。
    7. **调用者拆除**：弹出返回值和参数。
- 知识点：
  - 栈的构建和拆除是一个对称的过程，确保内存的正确分配和释放。
  - 这些步骤在底层由汇编指令实现。

7. 示例函数调用与汇编实现

- c语言代码示例:

```c
int Volta(int q, int r){
  int k;
  int m;
  ...
  return k;
}
int Watt(int a){
  int w;
  ...
  w = Volta(w,10);
  ...
  return w;
}
```

- 汇编实现:

1. **调用者准备**：将参数压入栈中 + **控制转移**：跳转到被调用函数 (JSR/JSRR)

```
Watt
  ; Push second arg 10
  AND R0,R0,#0
  ADD R0,R0,#10
  ADD R6,R6,#-1
  STR R0,R6,#0
  ; Push first arg
  LDR R0,R5,#-1 ; R0 <- w
  ADD R6,R6,#-1
  STR R0,R6,#0
  ; Call Subroutin
  JSR Volta
```

此时栈的结构如下:

```
高地址
|       |
|   w   | <- R6 (栈顶)
|  10   |
|       |
低地址
```

2. **被调用者准备**：Volta 函数开始执行时，需要构建自己的激活记录，压入书签信息和局部变量。

```
Volta:
    ; 留出返回值空间
    ADD R6, R6, #-1
```

```
    ; 压入返回地址
    ADD R6, R6, #-1
    STR R7, R6, #0

    ; 压入动态链接 (调用者的帧指针)
    ADD R6, R6, #-1
    STR R5, R6, #0

    ; 设置新的帧指针
    ADD R5, R6, #-1

    ; 为局部变量分配空间 (k 和 m)
    ADD R6, R6, #-2
```

- 逐步解析：
  1. 留出返回值空间：
     - `ADD R6, R6, #-1`：栈指针下移，为返回值预留一个位置。
     - 栈顶现在是空的，等待 Volta 执行完成后填充返回值。
  2. 保存返回地址：
     - `ADD R6, R6, #-1`：栈指针下移。
     - `STR R7, R6, #0`：将 `R7`（保存的返回地址）压入栈。
     - 这是为了在函数返回时能跳回调用者。
  3. 保存调用者的帧指针（动态链接）：
     - `ADD R6, R6, #-1`：栈指针下移。
     - `STR R5, R6, #0`：将调用者的 `R5`（Watt 的帧指针）保存到栈中。
     - 动态链接允许恢复调用者的栈帧。
  4. 设置新的帧指针：
     - `ADD R5, R6, #-1`：将 `R5` 设置为当前栈顶下方的一个位置，作为 Volta 的帧指针。
     - `R5` 现在指向激活记录的“基准”，便于访问参数和局部变量。
  5. 分配局部变量空间：
     - `ADD R6, R6, #-2`：栈指针下移两个位置，为 `k` 和 `m` 分配空间。
     - 在 LC-3 中，每个变量占用一个内存单元。

此时栈结构如下：

```
高地址
|       |
|   w   | <- 参数 q
|  10   | <- 参数 r
|   ?   | <- 返回值 (未填充)(可以用R5+2来访问)
|  R7   | <- 返回地址(可以用 R5+1来访问)
|  R5   | <- 调用者的帧指针 <- R5 (新的帧指针)
|   k   | <- 局部变量
|   m   | <- 局部变量 <- R6 (栈顶)
|       |
低地址
```

3. **函数执行，填充返回值**
4. **被调用者拆除 + 返回调用者**

```
; 将 k 复制到返回值位置
LDR R0, R5, #-1      ; 加载 k (返回值) (k在R5-1位置)
STR R0, R5, #2      ; 存储到返回值位置 (R5 + 2)

; 弹出局部变量
ADD R6, R5, #0      ; 将 R6 移到调用者帧指针处

; 恢复动态链接 (调用者的帧指针)
LDR R5, R6, #0
ADD R6, R6, #1      ; R6移动到返回地址处

; 恢复返回地址
LDR R7, R6, #0
ADD R6, R6, #1

; 返回控制权
RET
```

- 逐步解析：
  1. 更新返回值：
     - `LDR R0, R5, #0`：假设 k 存储在 R5 偏移 0 的位置（局部变量区），加载到 R0。
     - `STR R0, R5, #3`：将 R0 的值写入返回值位置（R5 + 3，即参数上方）。
     - 偏移量 3 是因为栈结构中返回值在参数和书签信息之后。
  2. 弹出局部变量：
     - `ADD R6, R5, #1`：将栈指针 R6 移动到调用者帧指针位置上方，跳过 k 和 m。
     - 这实际上“丢弃”了局部变量的空间。
  3. 恢复调用者的帧指针：
     - `LDR R5, R6, #0`：从栈顶加载调用者的 R5。
     - `ADD R6, R6, #1`：栈指针上移，完成弹出。

4. 恢复返回地址：
    - `LDR R7, R6, #0`：从栈顶加载返回地址到 R7。
    - `ADD R6, R6, #1`：栈指针上移。
5. 返回控制权：
    - `RET`：等价于 `JMP R7`，跳转到 R7 中的地址（Watt 的下一条指令）。

结束后的栈结构：

```
高地址
|        |
|   w    |
|  10    |
|   k    | <- 返回值已填充 <- R6 (栈顶)
|        |
低地址
```

5. **调用者拆除**：

```
; 获取返回值
LDR R1, R6, #0      ; 将返回值加载到 R1 (w)
ADD R6, R6, #3      ; 弹出返回值和两个参数
```

R6 恢复到调用`Volta`之前的位置，栈帧完全清理。

8. 为什么使用运行时栈?

- 选项1：为每个激活记录分配固定内存位置。
    - 问题：不支持递归，因为同一函数的多次调用会覆盖内存。
- 选项2：使用运行时栈。
    - 优点：每次函数调用都有独立的空间，支持递归。
- 知识点：
    - 运行时栈的动态性解决了固定内存分配的局限性。
    - 递归函数依赖栈的这种特性来保存每次调用的状态。

9. 为什么要用栈指针?

什么是帧指针?

帧指针（Frame Pointer，简称 FP，通常在 LC-3 中用 R5 表示）是一个特殊的寄存器，它指向当前函数的 **激活记录（Activation Record 或 Stack Frame）** 的某个固定位置。激活记录是函数调用时在运行时栈上分配的一块内存，用于存储函数的参数、局部变量、返回地址等信息。

简单来说：

- **栈指针（Stack Pointer，SP，LC-3 中用 R6）**：指向栈顶，动态变化（压栈时减小，弹栈时增大）。
- **帧指针（FP）**：指向当前函数栈帧的一个“基准点”，在函数执行期间保持不变，方便访问栈中的数据。

为什么需要帧指针?

栈指针（R6）会随着压栈和弹栈不断变化。如果只用栈指针来定位栈中的数据，会很麻烦，因为它的位置是动态的。而帧指针（R5）提供了一个固定的参考点，让我们可以轻松访问激活记录中的特定内容（比如参数或局部变量），即使栈顶在函数执行中发生了变化。

比喻

想象你在堆叠一堆书（运行时栈）：

- **栈指针（R6）** 是你当前操作的那本书（栈顶），你可能随时加书（压栈）或拿走书（弹栈）。
- **帧指针（R5）** 是你在一摞书中放的一个书签，标记某个特定位置（比如当前函数的起始点），无论上面加了多少书，你都能快速找到这个书签。

帧指针在运行时栈中的作用

在函数调用中，运行时栈会不断增长和缩减，尤其是在嵌套调用或递归时。帧指针的作用是：

1. **定位当前函数的数据**：提供一个稳定的基地址，方便访问参数、局部变量和返回值。
2. **保存调用关系**：通过保存上一个函数的帧指针（动态链接），支持栈的正确恢复。
3. **支持栈帧管理**：确保函数调用和返回时，内存分配和释放不会出错。

帧指针的具体好处

1. 稳定的参考点：
    - 比如在 Volta 中访问 q，可以用 `LDR R0, R5, #4`，而不用关心 R6 当前的具体位置。
2. 支持嵌套调用：
    - 如果 Volta 再调用另一个函数，R6 会继续下移，但 R5 保持指向 Volta 的栈帧不变。
3. 简化栈管理：
    - 弹栈时，只需将 R6 移到 R5 的位置，再逐步恢复，不用计算复杂的偏移。