Understanding the LC3 Runtime Stack

栈帧(Stack Frames)

栈帧包含函数在执行期间所需要的所有数据，至少包括：

- 给参数的空间
- 用于存放返回值、返回地址和其他书签信息的空间
- 局部变量的空间

栈帧在函数执行开始前创建，在函数执行完成后删除。因此，它满足仅仅在函数执行期间分配空间的要求。同时，它很好地处理了递归，因为每次调用都会创建一个新的栈帧。所有“数据”在物理上被集中在一起，这有助于缓存。

从概念上讲，栈帧是一个C struct。它定义了一组在内存连续块中放置在一起的变量列表。给定一个指针到struct，每个元素通过`ptr->element`的符号进行引用。

运行时栈（概念理解）

运行时栈从字面意义上理解就是一个在函数调用时实时变化的栈，遵循先进后出的原则。当前正在执行的函数的栈帧位于栈顶。其调用者位于下一个栈帧中，以此类推。当函数开始运行的时候，一个新的栈帧被压入栈中。控制权移交到被调用的函数，当其终止之后，控制权再返回调用函数。

一种实现栈的方法时通过单链表。`head`指向栈顶，栈的每个元素包含一个`next`指针指向后面的元素

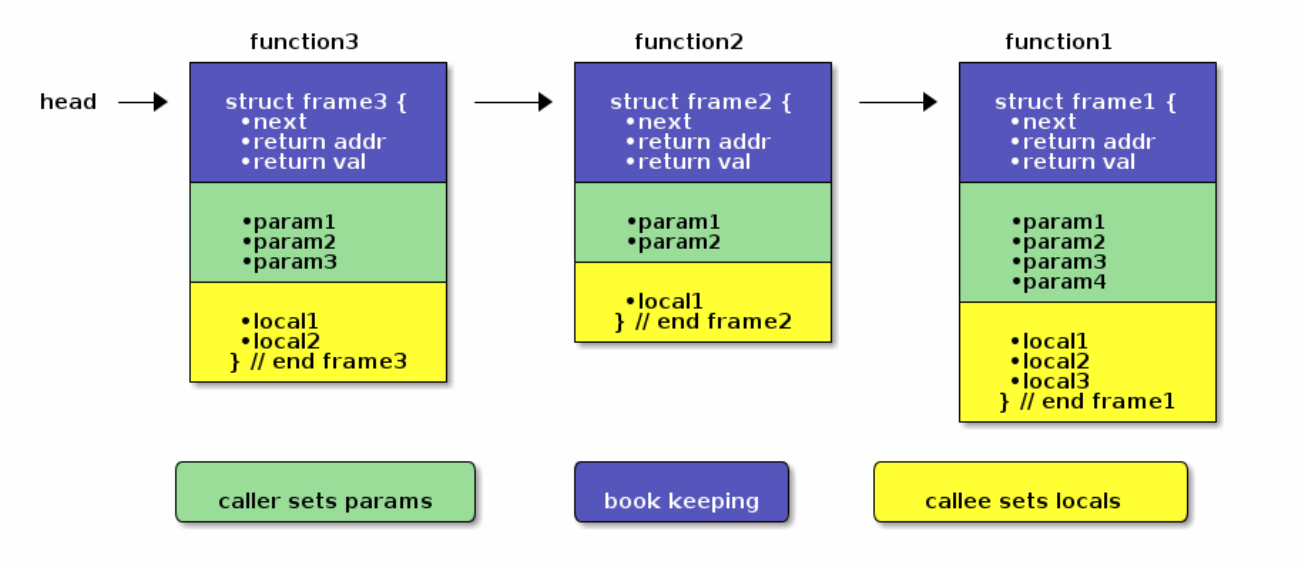
所以函数的“入口”看起来像是：

```
StackFrame* sf = malloc(sizeof(StackFrame));
sf->next = head; // link it into the list
head = sf; // reset the head
... // initialize parameters
```

而函数的“出口”看起来像是：

```
... // grad return value (if present)
StackFramd* sf = head;
head = sf->next; // reset the head(pop)
free(sf);
```

以图式形式，运行时栈看起来如下：



这表示三个嵌套的函数调用，当前正在执行的函数`function3`位于栈顶，并由`head`指向。你可以访问栈帧的所有字段（例如`head->param1`或`head->local1`）。通过`next`，你可以访问`function2`栈帧中的字段（例如`head->next->local1`）

虽然这个图和示例代码展示了正在发生的情况，但是存在几个问题：

- 栈帧的大小不同，那么空间该如何分配?
- 一个函数的栈帧大小可能在不同情况下不同，最简单的例子是`printf`，它有可变数量的参数
- 只用调用者知道传递的参数值，因此有了按值传递的概念。
- 只有函数本身知道它需要多少局部变量

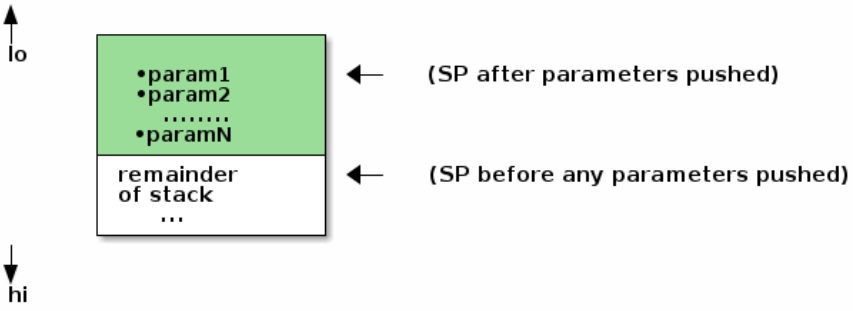
因此，构建活动记录需要调用者和被调用者之间的协作。并且动态分配空间（即`malloc`）对于函数调用来说开销太大。

构建LC3运行时栈

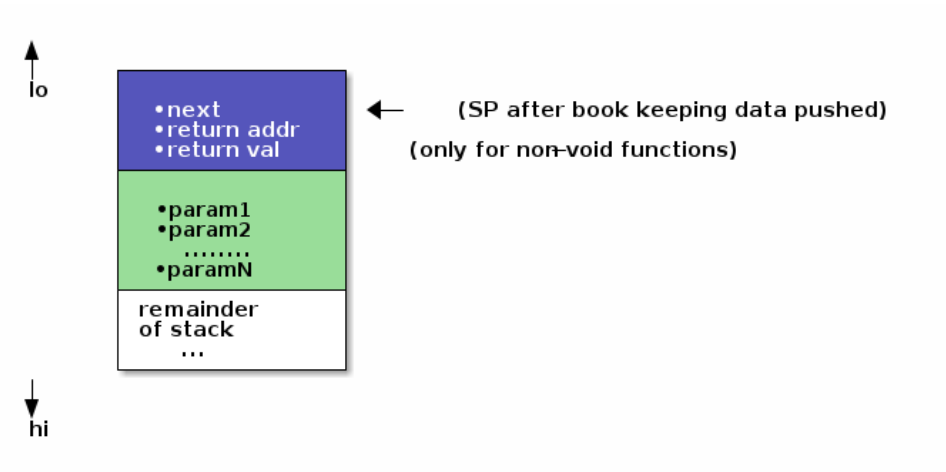
栈帧是由调用函数(caller)和被调用函数(callee)共同构建的。两个寄存器用于管理运行时栈。寄存器`R6`用作栈指针(stack pointer)，其值始终为栈顶的地址。寄存器`R5`包含栈帧链表的`head`指针，被称为帧指针(frame pointer)。

既然调用者知道参数的值，它会通过将它们逐个压入栈中，从而把参数传递给被调用的函数。这些值按照从右到左的顺序压入，因此，第一个参数最终位于栈顶。这样一来，无论实际传递了多少个参数，第一个参数都位于一个已知的位置。

函数调用开始时，调用者将参数从右到左压入栈中。结果是，在进入函数之前，栈看起来像这样：



然后代码使用JSR开始实行函数。现在轮到被调用函数接管。如果函数有返回值，它必须分配一些空间来存储它。通过将它放在栈上，返回时调用者可以通过简单地弹出它来获取该值。它还应该存储返回地址，以便这个函数可以调用其他函数。为了预留返回值的存储空间，函数只需调整栈指针（`ADD R6,R6,#-1`）。返回地址可以通过`PUSH R7`来保存。完成这两个步骤后，栈的状态如下：

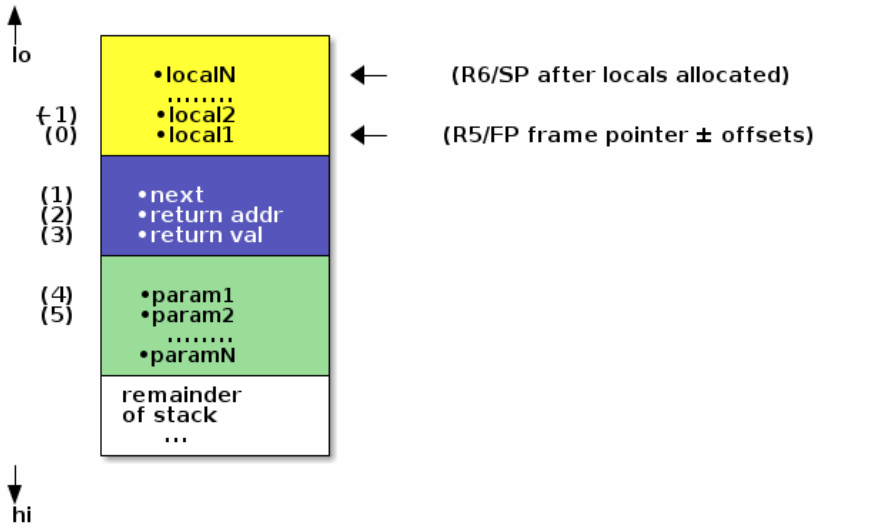


要完成栈帧的创建，需要两件事：

1. 将新的`stack frame`设为列表的新的`head`。
2. 为局部变量分配空间。

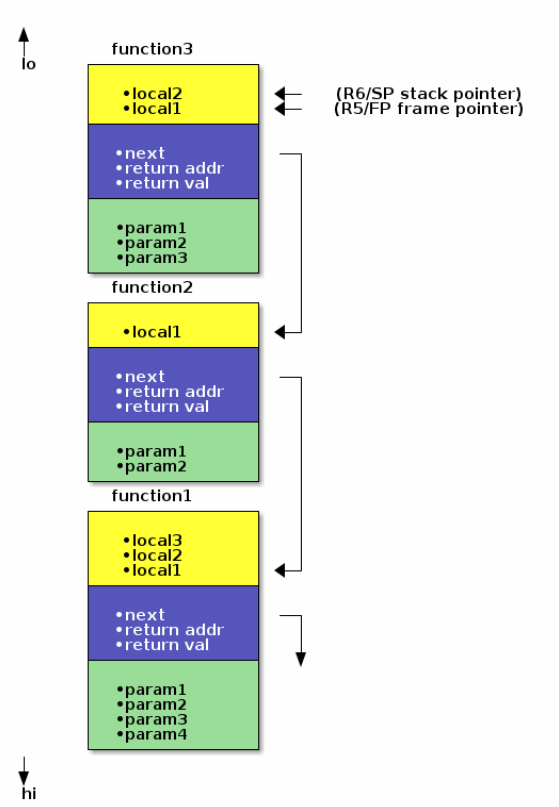
第一步需要将头部保存为新栈帧的`next`指针，这是通过`PUSH R5`实现的。然后，`head`被重置。`head`可以指向栈帧的任何位置。栈帧的元素通过相对于帧指针`R5`的固定偏移量来访问。由于LC3指令可以有正偏移和负偏移，因此帧指针指向栈帧中间位置来处理最大可能的栈帧是有意义的。所使用的约定是**头部（帧指针）指向第一个局部变量**。

最后，通过递减栈指针`R6`来分配局部变量的空间。此时，栈看起来像是这样：



带有三个嵌套函数调用的栈

注意三个函数在栈上是连续的。也就是说，被调用函数的最后一个参数紧邻调用函数的最后一个局部变量：



销毁运行时栈

在函数执行完成后，必须撤销构建运行时栈的工作。再次强调，这项工作是由被调用函数和调用函数共同完成的。撤销工作的顺序与构建栈的顺序相反。被调用函数清理其部分内容，然后调用函数在 `RET` 之后清理其部分内容。请注意，完成这项工作后，用于栈的内存不会被清除。它仍然包含之前存储的内容。然而，从逻辑上讲，它不再属于栈的一部分，并且在下一次函数调用时将被覆盖。

函数调用和return总结

使用运行时栈涉及很多步骤。但是，如果你严格按照步骤操作，这些步骤非常机械和直接。通常，代码是由编译器生成的。但是在该死的ECE220考试中，我们必须要手写这一坨屎：

1. 调用者从右到左压入参数
2. 调用者执行`JSR`
3. 被调用者为返回值预留空间
4. 被调用者压入返回地址
5. 被调用者压入旧的帧指针`R5`
6. 调用者重置帧指针（`ADD R5, R6 #-1`），这使得帧指针指向第一个局部变量。在步骤 10 中移除局部变量时，旧的帧指针位于栈顶，可以被弹出。
7. 调用者通过减栈指针（`ADD R6, R6, #-numLocals`）分配局部变量。
8. 调用者执行其工作（包括保存返回值 `STR Rx,R5,#3`）
9. 调用者通过增栈指针（`ADD R6, R6, #numLocals`）移除局部变量。
10. 调用者恢复旧的帧指针（`POP R5`）

11. 调用者弹出返回地址（`POP R7`）
12. 调用者执行`RET`
13. 调用者弹出返回值
14. 调用者通过增加栈指针来移除参数(`ADD R6, R6, #numParams`)
15. 调用者继续执行

以上为重点中的重点，务必牢记!