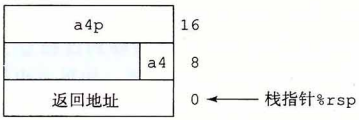

图 3-30　函数 proc 的栈帧结构。参数 a4 和 a4p 通过栈传递

练习题 3.33　C 函数 procprob 有 4 个参数 u、a、v 和 b，每个参数要么是一个有符号数，要么是一个指向有符号数的指针，这里的数大小不同。该函数的函数体如下：

```
*u += a;
*v += b;
return sizeof(a) + sizeof(b);
```

编译得到如下 x86-64 代码：

```
1   procprob:
2     movslq  %edi, %rdi
3     addq    %rdi, (%rdx)
4     addb    %sil, (%rcx)
5     movl    $6, %eax
6     ret
```

确定 4 个参数的合法顺序和类型。有两种正确答案。

3.7.4　栈上的局部存储

到目前为止我们看到的大多数过程示例都不需要超出寄存器大小的本地存储区域。不过有些时候，局部数据必须存放在内存中，常见的情况包括：

- 寄存器不足够存放所有的本地数据。
- 对一个局部变量使用地址运算符‘&’，因此必须能够为它产生一个地址。
- 某些局部变量是数组或结构，因此必须能够通过数组或结构引用被访问到。在描述数组和结构分配时，我们会讨论这个问题。

一般来说，过程通过减小栈指针在栈上分配空间。分配的结果作为栈帧的一部分，标号为“局部变量”，如图 3-25 所示。

来看一个处理地址运算符的例子，图 3-31a 中给出的两个函数。函数 swap_add 交换指针 xp 和 yp 指向的两个值，并返回这两个值的和。函数 caller 创建到局部变量 arg1 和 arg2 的指针，把它们传递给 swap_add。图 3-31b 展示了 caller 是如何用栈帧来实现这些局部变量的。caller 的代码开始的时候把栈指针减掉了 16；实际上这就是在栈上分配了 16 个字节。S 表示栈指针的值，可以看到这段代码计算 &arg2 为 $S+8$(第 5 行)，而 &arg1 为 S。因此可以推断局部变量 arg1 和 arg2 存放在栈帧中相对于栈指针偏移量为 0 和 8 的地方。当对 swap_add 的调用完成后，caller 的代码会从栈上取出这两个值(第 8～9 行)，计算它们的差，再乘以 swap_add 在寄存器%rax 中返回的值(第 10 行)。最后，该函数把栈指针加 16，释放栈帧(第 11 行)。通过这个例子可以看到，运行时栈提供了一种简单的、在需要时分配、函数完成时释放局部存储的机制。

如图 3-32 所示，函数 call_proc 是一个更复杂的例子，说明 x86-64 栈行为的一些特性。尽管这个例子有点儿长，但还是值得仔细研究。它给出了一个必须在栈上分配局部变量存储空间的函数，同时还要向有 8 个参数的函数 proc 传递值(图 3-29)。该函数创建一个栈帧，如图 3-33 所示。