第 8 讲书面作业包括两部分。第一部分为 Lecture08.pdf 中课后作业题目中的第 2、3 题。第二部分为以下题目：

A1. 以下是某简单语言的一段代码。语言中不包含数据类型的声明，所有变量的类型默认为整型（假设占用一个存储单元）。语句块的括号为‘begin’和‘end’组合；赋值号为‘:=’，不等号为‘<>’。每一个过程声明对应一个静态作用域（假定采用多遍扫描机制，在静态语义检查之前每个作用域中的所有表项均已生成）。该语言支持嵌套的过程声明，但只能定义无参过程，且没有返回值。过程活动记录中的控制信息包括静态链 SL，动态链 DL，以及返回地址 RA。程序的执行默认遵循静态作用域规则。

```
（1） var a0, b0, a2;
（2） procedure fun1 ;
（3）         var a1, b1;
（4）         procedure fun2 ;
（5）                 var a2;
（6）                 begin
（7）                         a2 := a1 + b1;
（8）                         if(a0 <> b0) then call fun3;
  .                           ......    /*不含任何 call 语句和声明语句*/
  .                  end;
  .          begin
                      a1 := a0 - b0;
                      b1 := a0 + b0;
（x）                 If   a1 < b1   then   call fun2 ;
  .                   ......      /*不含任何 call 语句和声明语句*/
  .          end ;
  . procedure fun3 ;
  .          var a3;
  .          begin
               a3 := a0*b0 ;
（y）                 if(a2 <> a3) call fun1 ;
```

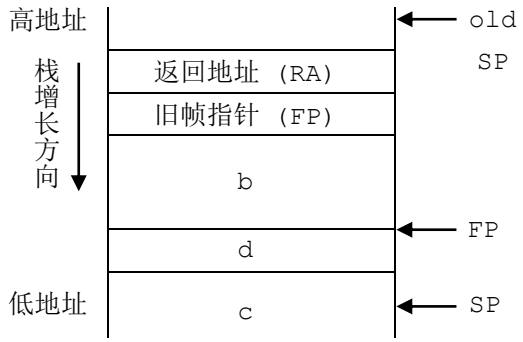
```
.                    ......    /*不含任何 call 语句和声明语句*/
.          end ;
.  begin
.          a0 := 1;
.          b0 := 2;
.          a2 := a0/b0 ;
.          call fun3;
.          ......    /*不含任何 call 语句和声明语句*/
.  end .
```

（a）当过程 fun2 被第二次激活时，运行栈上共有几个活动记录？依次是哪些过程的活动记录？当前位于次栈顶的活动记录中静态链 SL 和动态链 DL 分别指向什么位置？（注：指出是哪个活动记录的起始位置即可）

（b）若程序的执行改为遵循动态作用域规则， 则程序的执行会导致运行栈发生怎样的变化？

A2. 对于以下 C 函数片段，其运行时的活动记录按右图方式组织：

```
1)  static int N;
2)  void foo(int a) {
3)      short b[2020];
4)      int c[2 * N];
5)      unsigned long d = 996;
6)      c[2020] = a;
7)      b[1231] = 2019;
8)  }
```



其中 b, d 的大小是确定的，可直接确定其在栈内的偏移。而 c 是一个动态数组，编译器并不能确定将需要多少存储空间。这种组织方式将 c 存到 b, d 的下方（如图所示），用 FP 保存分配 c 之前的栈顶指针（SP），当分配完 c 时，再对栈顶指针做相应的调整（无需保存内情向量）。

设全局变量 N 存放的内存位置为 0x4000，参数 a 保存在寄存器 A0。一个 long 类型占 4 个字节，一个 int 类型占 4 个字节，一个 short 类型占 2 个字节。由该函数生成的某种 32 位计算机上的目标代码如下（如果不熟悉汇编指令，右侧给出了详细的注释帮助你理解）：

```
foo_prologue:
  addi sp, sp, ____①____   # SP <- SP + ??? (allocate new stack frame)
  sw   ra, ____②____(sp)   # M[SP + offset_RA] <- RA (store old RA)
  sw   fp, ____③____(sp)   # M[SP + offset_FP] <- FP (store old FP)
  mv   fp, sp              # FP <- SP (set new frame point)

foo_body:
  lw   t0, 0x4000(zero)# T0 <- M[0x4000] (load `N`)
  slli t0, t0, 0x3     # T0 <- T0 * 8
  sub  sp, sp, t0      # SP <- SP - T0 (allocate `c` on stack)

  li   t0, 996         # T0 <- 996
  sw   t0, ____④____(fp)   # M[FP + offset_d] <- T0 (d = 996)

  sw   a0, ____⑤____(sp)   # M[SP + offset_c] <- A0 (c[2020] = a)

  li   t0, 2019        # T0 <- 2019
  sh   t0, ____⑥____(fp)   # M[FP + offset_b] <- T0 (b[1231] = 2019)

foo_epilogue:
  _____⑦_____          # (deallocate `c`)
  _____⑧_____          # ???
  _____⑨_____          # ???
  _____⑩_____          # ???
  jr   ra              # PC <- RA (return)
```

回答以下问题：

(1) 试补全目标代码中缺失的偏移量，并参考函数调用起始阶段 foo_prologue 完成相应的函数调用收尾阶段 foo_epilogue（每空填一个数字或一条指令，意思正确即可，不需要考虑指令格式是否规范）。

(2) 源程序的第 6 行存在一个缓冲区溢出漏洞，即 N 取太小时会导致 c 数组访问越界，此时可能会覆盖掉栈上的一些数据。覆盖一般的数据影响不大，而如果被覆盖的数据恰好是函数返回地址，函数返回时就会跳转到错误的地址。通过精心构造该地址可使得程序执行流程发生更改，带来恶意代码执行等极其严重的后果。试问 N 取何值的时候恰好能覆盖该函数的返回地址？