目标代码生成

系统内存布局

如下图所示为一个 C 进程的虚拟内存布局，整体内存分为内核空间（Kernel Space）和用户空间（User Space），内核空间为操作系统占用，用户空间为用户程序可用空间。其中用户区域又进一步可分为堆栈区（Stack）、共享区（Shared Memory）、数据区（Data）和代码区（Text）。

一般内核空间在高地址区，用户空间从低地址区向高地址区延伸。用户空间的堆栈区紧邻内核区域，用户程序每创建一个线程，就会在堆栈区域分配固定大小的堆栈区域（一般约为 8MB）。每个函数运行时，系统都会为其在堆栈上分配一块规定大小的内存区域，用于存储传递给该函数的参数、函数的局部变量、计算的临时变量、函数返回后执行的下一条地址，以及保存的调用当前函数的函数运行时寄存器的内容等。处理器在运行时，有两个寄存器分别指向当前线程对应堆栈的不同位置，他们分别是 fp 指针和 sp 指针。sp 指针始终指向堆栈的栈顶，而 fp 指向当前运行函数的栈帧开始位置。编译器在编译程序时，能够计算出每个函数对应的栈帧的大小，并将对参数和局部变量等的访问转化为相对于 fp 和 sp 的相对偏移量，这样每个局部变量的存储地址在运行时可以根据调用序列动态确定，而无需在编译时给出。

如图所示有一个函数 func，该函数在 main()函数中调用后形成的堆栈情况如图中左上部分所示，栈底首先存储了一些很运行环境相关的内容（env），以及传递给 main 函数的参数(argv 和 argc)，紧接着是 main 函数的栈帧和 func 的栈帧。

```
func(int x, int y){
    int a;
    int b[3];
    ...
}
```

在图中右中给出了展开的该堆栈的内容，可以看出 func 的栈帧偏移量为 0 的单元存储了调用 func 的函数（main）的 fp 指针 mfp，接着是局部变量 a 和 b

的内容。由于 b 是一个 3 个元素的整型数组，假设整型占用 4 个字节，则整体占据了 16 个字节大小的空间。a 和 b 并未设定初始值，因此 a 和 b 对应的单元的值是随机的，程序直接读取会产生不确定的结果。

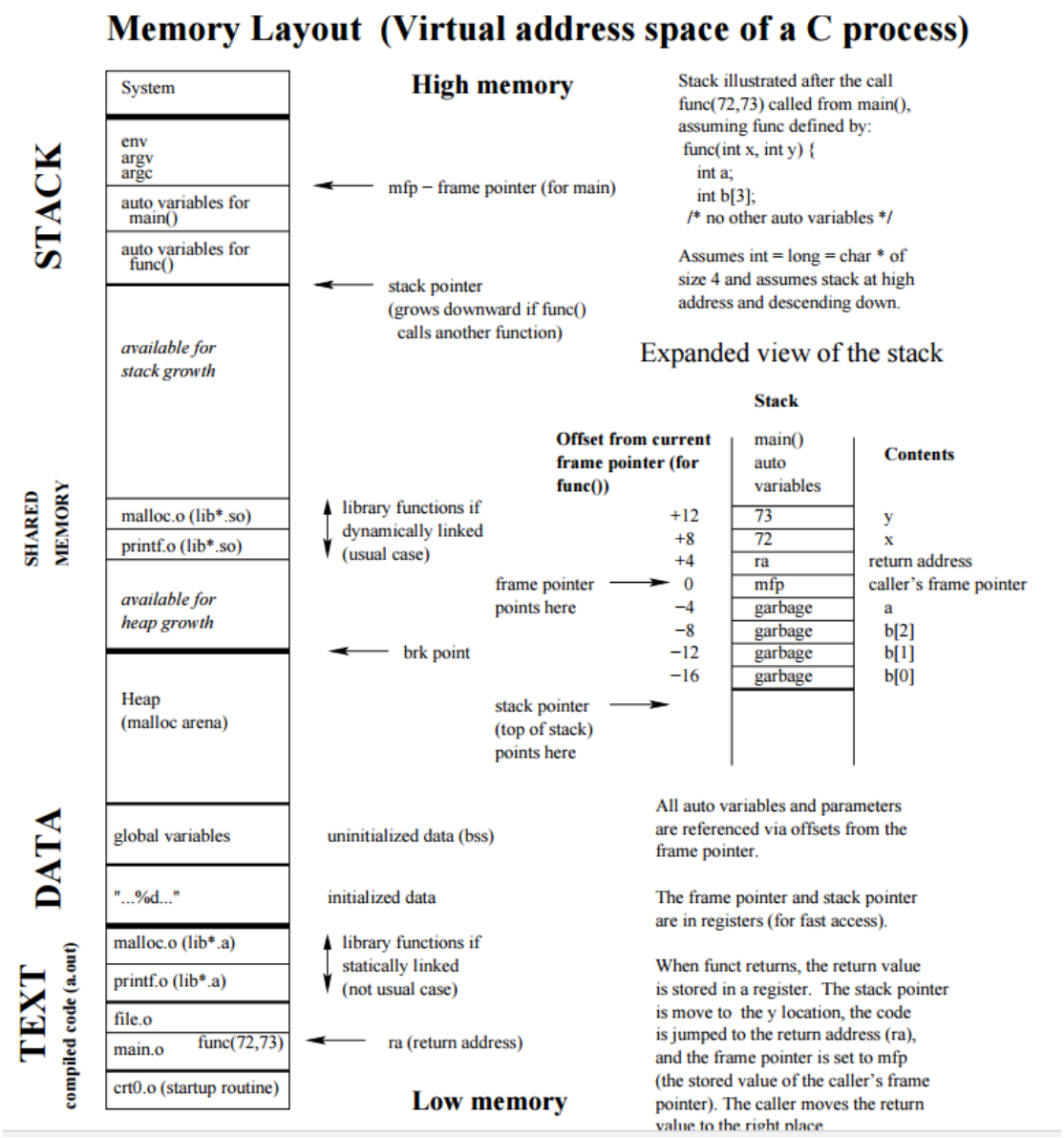
在图示的模型中，函数参数以及函数的返回地址是存储在调用函数的堆栈中的。例如 main 函数调用 func，传递给 func 的参数 x 和 y，以及 func 执行完后继续在 main 中执行的地址（ra）存储在 main 对应的栈帧中。堆栈是从高地址向低地址生长的，因此根据 sp 访问参数和返回地址时，偏移量为正数；而根据 fp 访问当前函数 func 的局部变量时，则偏移量为负数。

当函数返回时，func 函数的返回值一般放在一个寄存器中，sp 指针会指向变量 y 所在的位置，程序的 pc 寄存器会更新为 ra 单元的内容，fp 指针被更新为 mfp 单元的值，这样 func 对应的栈帧被释放掉，线程继续在 main 函数中执行。

进程之间共享的动态链接库在运行时加载在堆栈和堆区域之间。在 C 语言程序中，用户程序调用 malloc 等函数时，则会在堆区域分配一块内存，而调用 free 时，则会将该内存块释放掉。堆栈区域的分配按照先进后出的方式进行，分配和回收非常高效，而堆区域的分配则比较随机，经过多次的分配和释放之后，会造成比较多的内存碎片，因此内存管理的难度更大一些。

全局数据区包括已经初始化的区域（initialized data）和未初始化的区域（uninitialized data），全局变量分配在哪个区域，是由编译器根据分析的结果决定的。例如 C 语言里面的全局变量和静态变量都是分配在全局数据区。

代码区包括了用户编写的代码和一些静态链接的库程序，大小和每个应用程序有关，但是程序的入口地址一般是固定的，该区域内存储了编译之后的目标代码对应的指令序列，以函数为单位进行整体摆放。



实验目的

（1）了解编译器指令生成和寄存器分配的基本算法；

（2）掌握目标代码生成的相关技术和方法，设计并实现针对 x86/MIPS/RISC-V/ARM 的目标代码生成模块；

（3）掌握编译器从前端到后端各个模块的工作原理，目标代码生成模块与其他模块之间的交互过程。

实验内容

基于 BIT-MiniCC 构建目标代码生成模块，该模块能够基于中间代码选择合适的目标指令，进行寄存器分配，并生成相应平台汇编代码。

如果生成的是 MIPS 或者 RISC-V 汇编，则要求汇编代码能够在 BIT-MiniCC 集成的 MIPS 或者 RISC-V 模拟器中运行。需要注意的是，config.xml 的最后一个阶段“ncgen”的“skip”属性配置为“false”,"target"属性设置为“mips”、“x86”或者“riscv”中的一个。

如果生成的是 X86 汇编，则要求使用 X86 汇编器生成 exe 文件并运行。

实验过程与方法

基于上一次中间代码生成的基础，完成目标代码生成模块。

虽然指令选择和寄存器分配都存在多种算法，但是**本实验最重要的目标是能够生成目标代码，并在相应的系统平台（x86）或者模拟器（MIPS，RISC-V）中运行，走完编译器设计的整个流程**。各位同学可以根据中间代码以及前面模块的实现情况，选择适合于自己，且能够在时间节点之前完成的算法，有余力的同学可以选择课堂上讲到的更高级的算法生成更加高效的代码。

由于 RISC-V 模拟器实现的原因，RISC-V 带有乘除法的汇编无法运行。

386 汇编需要配置好 masm32。

对于内置的模块，BIT-MiniCC 提供了 8 个测试程序，**本次实验只要能通过其中任意 3 个测试程序可拿到 3 分，通过其中任意 4 个测试程序可拿到 4 分，通过其中任意 5 个及以上测试程序可拿到 5 分**。如有为了完成目标代码生成对前面模块的修正和改进的工作，请在实验报告部分给与说明。

如果发现代码抄袭，将按照 0 分计算。

关于 MIPS 处理器、RISC-V 处理器及汇编程序编写方法，可以从以下网址下载:

https://github.com/jiweixing/build-a-compiler-within-30-days

实验提交内容

本实验要求提交目标代码生成实现源码（项目 src 下所有文件），以及对应的 config.xml; C/C++需提供对应的可执行程序（不需要编译的中间文件），每个人提交一份实验报告。

实验报告放置在 doc 目录下，应包括如下内容:

- 实验目的和内容
- 实现的具体过程和步骤
- 运行效果截图
- 实验心得体会