2023.11.22 游震邦 ics茶话会

Overview

- Memory Safety
- Implementation of OOP

对CSAPP的批判

Goals of Programming Language

不可能三角

- safety：safety强调correctness，相比os和网络的security（机密信息不泄露）；python比c更safe c语言数组越界/错误的指针操作都能通过编译
- performance：性能
- productivity：程序员写起来快；学习曲线 学习成本 & 学会后写起来快

C performance最好，C++的performance差不多而safety和productivity更好，Java更好；python重在productivity
performance+productivity：Matlab
productivity+safety 静态类型Haskell：编译时会检查类型 C；相比运行时 动态 python
performance+safety：Rust

编程语言性能

1. C/C++, Rust
2. C#（和Java相似 但性能有时超过Fortran和Haskell）, Haskell

Safety

- Memory safety：和内存有关的各种问题
- Type safety：不会有和类型有关的错误 调用一个对象的方法时如果能通过编译就不会出错
- Thread safety
  编译器应该设计safety更好一些，而不需要程序员操心

Memory safety

Segmentation fault: 访问了不该访问的

1. invalid address
    1. null pointer dereference 空指针 地址名是无效地址
        1. java中最常见的exception
        2. null safety: 从类型安全上空指针不make sense；规定指针一定非空，空指针有另一种类型nullable type
    2. non zero
        1. cast a random object into a pointer
            1. 危险 类型系统弱的体现
        2. index out of bound / pointer arithmetic
            1. 除了c和rust不能做指针运算
2. improper right (protection) 权限错误
   1. write to read-only segment (eg. const；把main函数指针转化为int写 代码段只读)
   基本只有c语言和fortran有segmentation fault

Checking

- static 编译时
    - 通过内存系统rule out禁止做
      eg. cast a random object into a pointer
- dynamic 运行时
  eg. 数组越界是运行时检查 越界则抛出异常

Garbage collection

- Malloc/free
  - python java有garbage collector，不需要操心malloc是否被free; c不是
- 变量的Lifetime生命周期
  - static 生命周期和整个程序相同长
    - global/static variable 程序运行时放在固定位置
    - dynamic
      - 为什么需要dynamic：递归函数调用栈 每个栈帧中相同名称的变量有不同值
      - 栈上 stack
        - c++11 `auto` autonomously managed by compiler 变量的分配和释放自动由编译器管理（通过栈指针的加减）
        - 尽可能放在栈上而非堆：局部性好；不会出现memory leak
      - 堆上 heap
        - new delete
        - 把任意变量放在堆中任意位置
        - 为什么需要heap：
          - object live outside of its scope 一个变量的生命周期不能超过分配它的栈帧的生命周期——但不够本质，返回值可以拷贝出去，何时不能拷贝？
          - object too large：拷贝费时间&栈帧stack overflow（限制栈帧大小 为了safety防止无穷递归）
          - object of variable length
- **Memory leak**
  - mark & sweep
    - def
      - 变量之间有相互引用关系 停下程序 删去不可达的对象
      - 深度/广度优先遍历 能遍历到的则mark
    - cons
      - Time: P99 latency
      - Space: 2x 堆空间开销是实际两倍
    - pros
      - 多线程程序：不可检测c++是否多线程（rust可检测）；reference counting需要做atomic instruction
        - atomic instruction：counter的加减是atomic的加减；一个变量释放时它指向的变量也递归释放
  - reference counting
    - c++, rust, objective c, swift, lean, fortran
    - idea
      - no free, just malloc 堆上的变量和栈上的变量绑定
      - single ownership：unique ptr
        - vector 在堆上；析构时数组自动释放 堆上的vector底层数组和栈上的lifetime绑定
        - ownership：vector的数组的所有权是栈上的 栈上释放则底层数组释放
      - multiple ownership：shared ptr / weak ptr

CSAPP高度依赖C语言 尤其链接

OOP

- linux 70%是内存安全相关 现在有人用rust重写linux driver部分 假设linux不是c语言而是用现代语言写 许多内存安全都能解决
- 面向过程procedure：学sys到后面 每写下一行代码都能感受时间和空间开销（知道generate出的汇编代码）
- OOP中和性能有关的部分
  - OOP与c语言相比慢在何处?
    1. dynamic dispatch
    2. dynamic type

Polymorphism 多态

同名函数可以被作用到不同类型的实参上

- Generics
  - Static dispatch 流水线很深时预测下一个指令地址
- Virtual function
  - Dynamic dispatch 查表 唯一OOP无法优化掉的 调用开销巨大

Conclusion

计算机系统核心基石：编程语言+四大礼包
体会系统人会如何思考问题
performance engineering

系统人如何选择学习编程语言?

- 可参考https://github.com/ZhenbangYou/Computer-Systems-Learning-Resources-A-Recommended-List
- 函数式编程 functional programming
- Kotlin
- F#
- Scala
- Rust
- C++ 强烈建议学20
- extra
  - C#
  - Swift

学编程语言，除工作需求外主要是兴趣
Rust和Haskell是最难学的语言之二