CSC2/452 Computer Organization
Mixed Language Programming

Sreepathi Pai

URCS

December 7, 2022

Outline

Outline

Administrivia

- A5 (final assignment) is out
  - Due Dec 13, 2022 at 7PM
  - One more interesting bug before whole class gets extra credit
- All homework grades and solutions available
  - Review them and ask questions on Blackboard
- Two review lectures next week
- Exam will be 90–120 minutes, one sheet of handwritten notes allowed

Outline

Reason #1: Reuse

- Reuse
  - The best way to write bug free code is not write code at all
  - The second best way is to reuse existing, well-tested, already available code
- Example: Numpy
  - `numpy`: Numerical programming for Python
  - Matrix manipulations, multiplication, etc.
  - Much code reused from earlier projects like BLAS

Reason #2: Performance

- Performance
  - Many languages focus on productivity, not raw performance
  - 80% of the code doesn't need performance (e.g. UI, File I/O)
- Find hotspots in your code and write them in language capable of delivering high performance
  - Using a tool called a 'profiler'
- Example: Numpy
  - when you multiply matrices in Numpy, it almost inevitably calls `gemm` in BLAS

Reason #3: Accelerators/Machine capabilities

- New instructions and extensions
  - "Advanced Matrix Extension" (AMX) Intel's matrix multiply instruction
  - "Advanced Vector Extensions" (AVX) for vector/SIMD processing
  - "Multimedia Extensions" (MMX) for video processing
- New accelerators
  - GPUs require you to write code in CUDA or OpenCL
  - Shouldn't have to rewrite entire application in CUDA/OpenCL

Outline

Native Code

- Languages that compile to native code (i.e. machine code/assembly language)
  - C++ program calls C functions (or vice versa)
  - Rust programs calls C functions
  - C calls CUDA
- Mostly for reuse or to access machine capabilities
- Relatively easy:
  - The programs ultimately compile to assembly language
  - Just compile them individually and link them
- Interoperability must be ensured:
  - Each language must follow the ABI
  - Each language has its own rules (naming, calling conventions, etc.)
  - Each language has its own data formats

Example: C++ calling C

Contents of `test.cpp` file.

```
#include <cstdio>

int add2num(int a, int b);

int main(int argc, char *argv[]) {
  int c, d;

  c = 3;
  d = 4;

  c = add2num(c, d);
}
```

Contents of `add2num.c` file.

```
int add2num(int a, int b) {
  return a + b;
}
```

Compiling and Linking

```
g++ -c test.cpp -o test.o
gcc -c add2num.c -o add2num.o
gcc test.o -o mlp_compiled
test.o: In function ‘main’:
test.cpp:(.text+0x28): undefined reference to ‘add2num(int, int)’
collect2: error: ld returned 1 exit status
make: *** [mlp_compiled] Error 1
```

Name Mangling in C++

C++ supports polymorphism, so function names are `mangled` at the assembly level.
Here is the name the C++ program is looking for (generated by C++ compiler):

```
objdump -t test.o
[...]

0000000000000000         *UND*    0000000000000000 _Z7add2numii
```

Here is the name the C compiler generated:

```
objdump -t add2num.o
[...]
0000000000000000 g     F .text    0000000000000014 add2num
```

Handling name mangling

```
#include <cstdio>

extern "C" int add2num(int a, int b);

int main(int argc, char *argv[]) {
  int c, d;

  c = 3;
  d = 4;

  c = add2num(c, d);
}
```

Now, recompilation works

```
gcc -c add2num.c -o add2num.o
g++ -c test2.cpp -o test2.o
gcc add2num.o test2.o -o mlp_compiled_2
```

Bytecode Languages: The easy cases

- Usually, but not always, interpreted
  - They don't compile to assembly
- Byte code programs calling each other
  - Usually all need to run on top of the same VM/interpreter
  - Examples: IronPython and C# (.NET)
  - Or: Jython and Java (JVM)
  - Logically similar to native languages, but more "natural" since all languages follow the VM rules
  - Interesting combination: JavaScript calling WebAssembly (both are bytecode)

Bytecode Languages: The hard cases

- Byte code calling Native code
  - Python calling C
  - Java calling C
- Native code calling byte code
  - C calling Python functions
- These almost always need “marshalling”
  - Converting data types between VM and native representations
  - Transferring control between native and byte code/VM code correctly

Outline

Using asm in C

```
#include <stdio.h>

int add2num(int a, int b) {
  asm("addl %1, %0" : "+r" (a) : "r" (b));
  return a;
}

int main(void) {
  printf("%d\n", add2num(1, 3));
}
```

- asm keyword contains assembly opcode
- Also contains argument references %0
- Also contains information on how to pass arguments to assembly instruction
  - Here r means register
  - The + indicates the register will be modified

Alternatives to writing `asm`

- Use assembly-language intrinsics
  - Basically assembly language dressed up to look like C functions
  - Example: `_mm256_mul_pd` corresponds to the `VMULPD` instruction
- Most commonly for SIMD functionality
- Look exactly like C function calls, except they may use special data types
  - For `_mm256` intrinsics, 256-bit values

Outline

Using ctypes

- This assumes the C code you want to call is a shared object (or a DLL).
- The code doesn't have to be aware of Python's rules

```
#!/usr/bin/env python3

import ctypes

libadd2num = ctypes.cdll.LoadLibrary("add2num.so")

libadd2num.add2num.argtypes = [ctypes.c_int, ctypes.c_int]
libadd2num.add2num.restype = ctypes.c_int

print(libadd2num.add2num(1, 3))
```

- ctypes does the hard work of translating types from Python to C and vice versa under the hood.
- Another alternative is to use c_ffi library

Outline

Writing Modules in C

- Many modules (e.g. ‘import x‘) are written in C
- They’re much more closely integrated with Python
- Called “Extension Modules”
  - See the documentation here: `https://docs.python.org/3/extending/extending.html`

Writing Modules in Cython

- Cython allows you to write native code modules in a Python-like language
- Cython compiles these to Python modules
- Very convenient if you want to speed up code with less effort than writing in C
- Recommended `https://cython.readthedocs.io/en/latest/src/tutorial/cython_tutorial.html`

Outline

Pure X programming is portable

- "Pure Python Programs"
  - Only need the Python interpreter to work
  - CPython, PyPy, Jython, IronPython, etc.
- Python + C modules dramatically cuts portability
  - tied to Cython and C

Mixed language programming can be brittle

- All languages need to coordinate
- Many languages provide "foreign function interfaces" to make this easier
- May need to understand *all* languages at a fairly low level
- Intellectually rewarding
  - Not very hard since ultimately everything is assembly