Навыки связи библиотек Python с программами на C/C++: PyBind, Cython, Ctypes

Сергей Матвеев, Сергей Рыкованов

Plan for today

1. Reminder about Linux and monolitic kernel
2. Reminder on Static/Dynamic linking;
3. Tools for acceleration of Python code;
   - os $\rightarrow$ os.system(somebin) :)
   - `ctypes` call C from Python;
   - Numba;
   - Cython;
   - Pybind11 call Python from C++, call C++ from Python!
   - Run snippets.

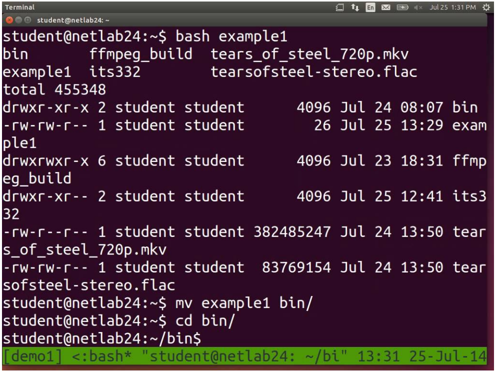
The Operating System and the User

The OS provides the User interaction with the "Programs"(processes) by means of

- Graphical Interface. Native for Windows, possible for UNIX/Linux OS: KDE, Gnome, XFce, etc.
- Command Line interface. Native for UNIX/Linux OS, Windows simulates it (power shell).
- Application Programming Interface (API) provided by OS Kernel (Interprocesses interactions, pipes, sockets), C/C++ compiler is needed. For advanced users and developers.

It is completely enough the command line interface for administration purposes.

The Command Line (shell)



We can easily manage and "interact with a computer"by around two dozens of commands like **ls**, **cd**, **mkdir**, **pwd**, **cat**, **date**, **whoami**, **su**...

But what is beyond the "user-friendly commands"?

– The main functions of the Operating system –

Operating System (OS) and its main functions

Operating system is a set of programs controlling

- Existence
- Usage
- Distribution

of the computing system resources.

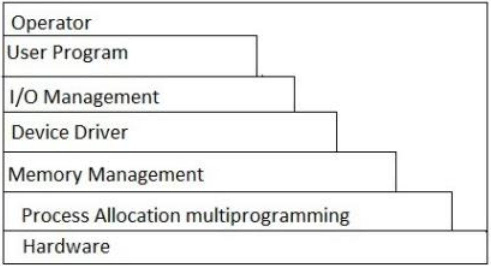


fig:- layered Architecture

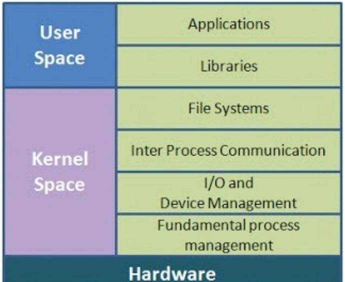
Each operating system works with basic its units: processes

OS kernel

Kernel is a part of OS working **permanently** in RAM, in **privileged regime** processing control of

- Process activities scheduling
- Interaction of processes
- Memory distribution
- Hardware and software drivers
- I/O and filesystem management

All of this functions are provided by **one** kernel program in case of **monolithic** kernel – Unix/Linux are organized exactly in this way.



Process

Process is a set of commands and data processes by computer having permissions to use computing resources. OS manages implementation of scheduling and interaction between many processes inside single computer. Each process goes though several stages during its **lifecycle**:

1. Generation of process
2. Execution at CPU
3. Pending for allocation of resources and interations with other processes
4. Completion of process and release of allocated resources

Process context

Context of process is data characterizing its current state. Consists from

- User context (commands, code segment and data)
- System context (id, permissions, opened files and i/o sstreams etc)
- Hardware context (e.g. registers and setup at stopping point)

At any one time during execution a **process runs in the context of itself** and the **kernel runs in the context of the currently** running process.

Process and Thread are different.

Filesystem basic concepts

Filesystem is a part of OS consisting from

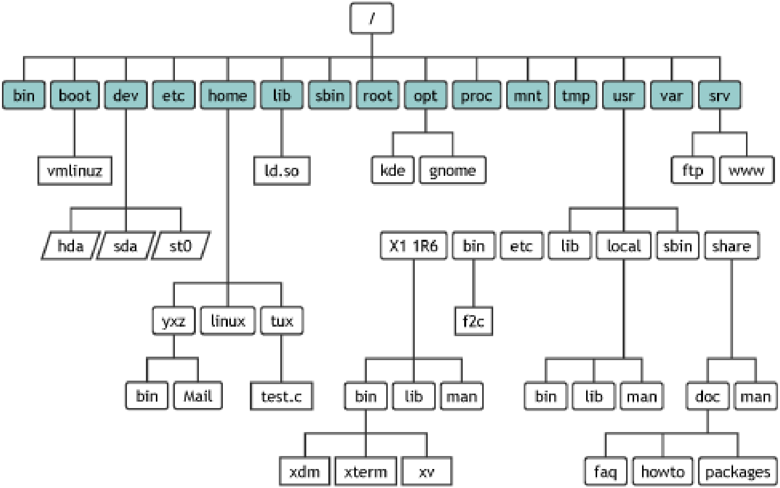
- Organized datasets
- Stored at external devices
- and **software** tools guaranteeing name-addressed access to these datasets and their protection

Thus, **file** is just series of bytes associated with **attributes**

File attributes

- Name – just set of symbols
- Acess perimssions
- Personification (stores info about file's "owner" and "usergroup")
- Type of file (there are files for devices and regular files; files for execution and data)
- Block size
- Size of file
- Read/Write pointer
- Etc (e.g. last modification time)

Linux filesystem



Short plan

1. Reminder about Linux and monolitic kernel
2. Reminder on Static/Dynamic linking;
3. Tools for acceleration of Python code;
    - os $\rightarrow$ os.system(somebin) :)
    - `ctypes` call C from Python;
    - Numba;
    - Cython;
    - Pybind11 call Python from C++, call C++ from Python!
    - Run snippets.

About Python

Great advantages:

- Easy to learn
- Easy to get complicated libraries
- Easy to combine purpuses

Great minuses:

- Slow
- Interepreter causes troubles with shared-memory parallelism
- Runtime errors...

About C/C++

Great advantages:

- Fast
- Compile once and use long
- OpenMP

Great minuses:

- Study for all life
- SegFault every time :)
- Libraries require some skills

Big Dream.

- Easy to learn
- Easy to get complicated libraries
- Easy to combine purpuses
- Fast
- Compile once and use long
- OpenMP or smth like this

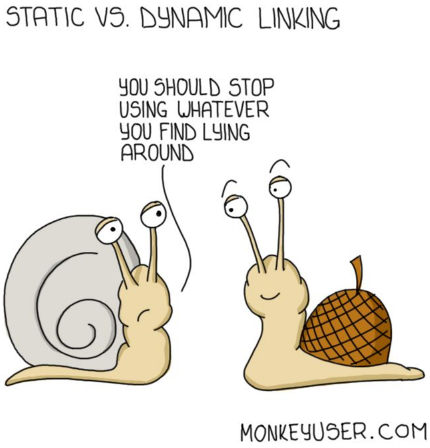
When it is a good idea?

- Need to accelerate particular parts of code
  E.g. a lot of work with matrices/tensors or many for-type loops
- Use of really tuned and large libraries
  NLopt, openGL, openCV
- os.system call binary – why not?

Compiler

- Check the compiler and its version:
  which gcc
  gcc −− version
- Prepare the object files:
  -c Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file

Dynamic and Static



Ok. Let's do it

```
gcc -c liblib.c
gcc -c liblib2.c
gcc -c main.c
gcc main.c liblib.c liblib2.c -o EXAMPLE.exe
gcc main.c liblib.o liblib2.o main.o -o EXAMPLE.exe
```

```
main.o: In function ‘main’:
main.c:(.text+0x0): multiple definition of ‘main’
/tmp/ccp6FWPX.o:main.c:(.text+0x0):
first defined here
collect2: error: ld returned 1 exit status
```

FAIL

ld

Our superhero here is **ld** – linker!
The corresponding environment variables are LD_LIBRARY_PATH LIBRARY_PATH

Ok. Next try

```
gcc -c liblib.c
gcc -c liblib2.c
gcc -c main.c
gcc main.c liblib.o liblib2.o -o EXAMPLE.exe
```

BETTER

Ok. Use ar to combine objectives

```
gcc -c liblib.c
gcc -c liblib2.c
gcc -c main.c
ar rc minilib.a liblib.o
gcc main.c minilib.a -o EXAMPLE.exe
gcc main2.c minilib.a -o EXAMPLE2.exe
```

BEST

minilib.a is a static library!

And don't forget optimization or debug!

```
gcc -c liblib.c -O3
gcc -c liblib2.c -g
gcc -c main.c
ar rc minilib.a liblib.o
gcc main.c minilib.a -o EXAMPLE.exe
gcc main2.c minilib.a -o EXAMPLE2.exe
```

gdb – is GNU debugger for C/C++/Fortran (when printf is not enough for debug)

Dynamic linking

```
gcc -c -Wall -Werror -fpic foo.c
gcc -shared -o libfoo.so foo.o
gcc -Wall -o test main.c -lfoo
/usr/bin/ld: cannot find -lfoo
collect2: ld returned 1 exit status
gcc -L$PWD/foo  -o test main.c -lfoo
echo $LD_LIBRARY_PATH
gcc -L$PWD/foo -Wl,-rpath=/home/username/foo
      -o test main.c -lfoo
```

* PIC – Position Independent Code

Thee Pillars of Makefile

target: dependencies
[tab] command

Data conversion

Main idea: Python int != C/C++ int

type	C/C++	Python
int	Fixed size	Arbitrary size[1]
float	Fixed precision	Arbitrary precision
complex	Built-in (but no built-in conversion)	Built-in
string	Built-in (but no built-in conversion)	Built-in
bool	Built-in (built-in conversion)	Built-in

[1] example here!

Our target Python procedure

```
def monte_carlo_pi_for(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Ok, in C it looks like this...

```c
double compute_pi(int nsamples)
{
    int i;
    int acc = 0;
    srand(time(NULL));
    double x, y, z;
    for (i = 0; i < nsamples; i++)
    {
      x = (double)rand() / RAND_MAX;
      y = (double)rand() / RAND_MAX;
      z = x * x + y * y;
      if (z <= 1)
          acc++;
    }
    return (4.0 * acc) / nsamples;
}
```

gcc -fPIC -fopenmp -O3 -shared -o libPI.so compute_pi.c

Ok, let us do binding with Ctypes

```
import ctypes #
clibPI = ctypes.CDLL('./libPI.so') #
n = 10000000
answer = clibPI.compute_pi(n)
```

Ok, let us do binding with Ctypes

```
import ctypes #

clibPI = ctypes.CDLL('./libPI.so') #
n = 10000000
answer = clibPI.compute_pi(ctypes.c_int(n)) #
```

Ok, let us do binding with Ctypes

```
import ctypes #

clibPI = ctypes.CDLL('./libPI.so') #
clibPI.compute_pi.restype = ctypes.c_double #
n = 10000000
answer = clibPI.compute_pi(ctypes.c_int(n)) #
```

Easy path to Openmp now!

```
double par_for(int nsamples)
{
    double x = 0.0;
    #pragma omp parallel for reduction(+:x)
    for (int i = 0; i < nsamples; i++)
    {
        if (i % 2 == 0)
            x += i * 0.5;
    }
    return x;
}
```

Easy path to Openmp now! (where is the bottleneck???)

```c
double compute_pi(int nsamples)
{
  int i;
  int acc = 0;
  srand(time(NULL));
  double x, y, z;

#pragma omp parallel for reduction (+:acc) private(x,y
  for (i = 0; i < nsamples; i++)
  {
    x = (double)rand() / RAND_MAX;
    y = (double)rand() / RAND_MAX;
    z = x * x + y * y;
    if (z <= 1)
        acc++;
  }
  return (4.0 * acc) / nsamples;
}
```

```python
@jit(nopython=True)
def jit_monte_carlo_pi_for(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
```

```python
from numba import njit
@njit
def f(n):
    s = 0.
    for i in range(n):
        s += sqrt(i)
    return s
```

Opportunities of numba

- Python functional known by numba and in particular
- Numpy functional known by numba [2]

More details

- Python lists
- Numpy arrays
- Tuples
- Dicts

What cannot be accelerated:

- pandas
- scipy
- many others

[2] habr pt 1

Useful options

There are also some more useful options[3]

- **nogil=True**
- **parallel=True**
- **cache=True**

[3] habr pt 2

Cython

- Types like in Python
- Definitions with C style
- Faster and cheaper way

example.pyx

```
from libc.math cimport pow

cdef double square_and_add (double x):
    """Compute x^2 + x as double."""
    return pow(x, 2.0) + x

cpdef print_result (double x):
    """This is a cpdef function
     that can be called from Python."""
    print("({} ^ 2) + {} = {}".format(x,
    x, square_and_add(x)))
```

setup.py

```python
from distutils.core import Extension, setup
from Cython.Build import cythonize
# define an extension that
#will be cythonized and compiled
ext = Extension(name="example",
         sources=["example.pyx"])
setup(ext_modules=cythonize(ext))
```

python setup.py build_ext inplace

myscript.py

```
import example
A = example.double square_and_add(10)
print(A)
```

Back to computations

```
import random
cpdef double compute_pi (int nsamples):
    """
    compute pi
    parameter: nsamples -- integer
    """
    cdef int i
    cdef double x, y
    cdef int acc
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x **2 + y**2 <= 1):
            acc = acc + 1
    return 4.0 * acc / nsamples
```

PyBind11

First steps

This sections demonstrates the basic features of pybind11. Before getting started, make sure that development environment is set up to compile the included set of test cases.

Compiling the test cases

Linux/MacOS

On Linux you'll need to install the **python-dev** or **python3-dev** packages as well as **cmake**. On Mac OS, the included python version works out of the box, but **cmake** must still be installed.

After installing the prerequisites, run

```
mkdir build
cd build
cmake ..
make check -j 4
```

The last line will both compile and run the tests.

Windows

On Windows, only **Visual Studio 2015** and newer are supported since pybind11 relies on various

Before we start

```
git clone https://github.com/pybind/pybind11.git
cd pybind11
mkdir build
cd build
cmake ..
make install
```

conda install pybind11 works

Basic example

```cpp
// hello.cpp
#include <pybind11/pybind11.h>
#include <iostream>
using namespace std;
int add(int i, int j) {
    cout << "Hello from C++!" << endl;
    return i + j;
}
PYBIND11_MODULE(hello_world, m) {
    m.doc() = "pybind11 hello world plugin";
    m.def("add", &add,
    "A function which adds two numbers");
}
```

g++ -O3 -Wall -shared -std=c++11 -fPIC ‘python3 -m pybind11 –includes‘ hello.cpp -o hello‘python3-config –extension-suffix‘

Now try how it looks like..

PyBind11

We are also able to use

- Python objects as variables for C++ functions (see daxpy example)
- Use Openmp inside calls
- Setup and call Python-functions with lambda-functions inside of C++ code
- Use standard Python calls inside of C++ code

Run snippets

PyBind11

And also advanced things

- Custom data structures
- Binding custom data structures
- OOP bindings
- Work with STL containers
- Advanced Numpy bindings

Hometask

- Реализовать на языке C/C++ классические операции перемножения квадратных матриц и умножения матрицы на вектор (15%);
- Разделить программу на несколько модулей и провести сборку через статическую линковку (25%);
- Подготовьте две сборки с флагами -g и -O3 и измерьте времена выполнения операций с $N = 512, 1024, \ldots, 4096$ (20%);
- Выполните вызов процедуры из Python через Ctypes/Cython/PyBind11 и измерьте времена (40%);

Бонусы:

- Дополнительные баллы за использование функций BLAS/cBLAS/openBLAS;
- Дополнительные баллы за вызов теста LINPACK на вашем компьютере;
- Супербонус: реализовать метод Штрассена перемножения квадратных матриц.