Einbetten von Python in C/C++ Programmen

Linuxwochen Eisenstadt, 23. 5. 2014

Agenda

1. Short introduction to Python
2. Single thread solution
3. Python objects
4. Multi threaded applications
5. Calling module functions

Python

- Alte Sprache: Seit Ende der 80 Jahre
- Guido van Rossum, CWI, Niederlande

- Aktuelle Versionen (stable):
  - 3.4.1
  - 2.7.6 ← wurde in diesem Projekt verwendet

Single thread – no problem

```
#include <stdio.h>
#include <Python.h>
int main(int argc, char * argv[])
{
  // initialize the interpreter
  Py_Initialize();
  // evaluate some code
  PyRun_SimpleString("import sys\n");
  // ignore line wrap on following line
  PyRun_SimpleString("sys.stdout.write('Hello from an embedded
Python Script\n')\n");
  // shut down the interpreter
  Py_Finalize();
  return 0;
}
```

Quelle: http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/036/3641/3641l1.html

How to call Python code?

- Simple functions:
  - `PyRun_SimpleString, PyRun_SimpleFile`
  - No return value possible
- With return value:
  - `PyRun_String, PyRun_File`
  - Return value of `PyObject *`

https://docs.python.org/2/c-api/veryhigh.html

PyObject – what???

- All kind of objects are PyObject
  - Object
  - Numeric objects
  - Sequence, Mapping
  - Files
  - Code
  - …

Creating objects

- Converting from C variable
  - `PyInt_FromLong, PyFloat_FromDouble, …`
  - `PyUnicode_FromStringAndSize, …`
- New objects with `PyXXX_New(…)`
  - `PyDict_New, PyList_New, …`
  - Filling with: `PyDict_SetItemString, PyList_SetItem`

Reading objects

- Converting to C variables
  - `PyInt_AsLong, PyFloat_AsDouble, …`
  - `PyString_AsString, PyString_Size, …`
    - Convert to UTF-8 before: `PyUnicode_AsUTF8String`
- Get PyObject from complex objects
  - `PyList_Size & PyList_GetItem, …`
  - Iterator interface: `PyDict_Next, …`

The reference counter

- Python objects are not copied, instead the reference counter is increased by 1.
- So, after using a variable call `Py_DECREF`
  - `Py_XDECREF` checks for NULL object before.
- Certain functions return a borrowed reference, NO `Py_DECREF` necessary.
  - API: *Return value: Borrowed reference.*

Checking objects

- Used to find the type of object
  - `int PyXXX_Check(PyObject *o)`
    - e. g. `PyDict_Check, PyString_Check, …`
- Also used for abstract protocols
  - Like the mapping protocol

Multi Threading makes things complicated

- Locking:
  - Global interpreter lock
  - Allows the Python interpreter to run all of its own threads
  - `PyEval_AcquireLock, PyEval_ReleaseLock`

- Thread State
  - Every thread of the calling application maintains its own state information in form of `PyThreadState` object

Main initialization

```
PyThreadState * mainThreadState = NULL;

void initMainThreadState()
{
  // Initialize python interpreter
  Py_Initialize();

  // Initialize thread support
  PyEval_InitThreads();

  // Save a pointer to the main PyThreadState object
  mainThreadState = PyThreadState_Get();

  // Release the lock
  PyEval_ReleaseLock();
}
```

Thread initialization

```
PyThreadState * threadState = NULL;

void initThreadState()
{
  // Get the global lock
  PyEval_AcquireLock();

  // Get a reference to the PyInterpreterState
  PyInterpreterState * mainInterpreterState = mainThreadState->interp;

  // create a thread state object for this thread
  threadState = PyThreadState_New(mainInterpreterState);

  // free the lock
  PyEval_ReleaseLock();
}
```

Perform calls to Python interpreter

```
// Grab lock
PyThreadState * tempState = pythonBegin();

// Executing Python code using
// PyRun_*, PyObject_CallObject, etc.

// Release lock
pythonEnd(tempState);
```

Helper functions

```
PyThreadState * pythonBegin()
{
    // Grab the global interpreter lock
    PyEval_AcquireLock();

    // Swap in my thread state
    return PyThreadState_Swap(threadState);
}

void pythonEnd(PyThreadState * tempState)
{
    // clear the thread state
    PyThreadState_Swap(tempState);

    // release our hold on the global interpreter
    PyEval_ReleaseLock();
}
```

Finalize thread

```
void finalizeThread()
{
  // Grab the lock
  PyEval_AcquireLock();

  // Swap thread state into the interpreter
  PyThreadState_Swap(threadState);

  // Perform clean-up on Python objects, etc.

  // Swap my thread state out of the interpreter
  PyThreadState_Swap(NULL);

  // Clear out any cruft from thread state object
  PyThreadState_Clear(threadState);

  // Delete my thread state object
  PyThreadState_Delete(threadState);

  // Release the lock
  PyEval_ReleaseLock();
}
```

Finalize Python interpreter

```
void finalizeMainThread()
{
  // Grab the global interpreter lock
  PyEval_AcquireLock();

  // Swap in main thread state
  PyThreadState_Swap(mainThreadState);

  // ... and finalize
  Py_Finalize();

  mainThreadState = NULL;
}
```

Function calls to modules

(without knowing the parameters)

```
// modulename as char *
PyObject * module = PyImport_ImportModule(modulename);

PyObject * dict = PyModule_GetDict(module);

// functionname as char *
PyObject * function = PyDict_GetItemString(dict,
  functionname);

// parameters as PyObject *, created with PyTuple_New
PyObject * returnValue = PyObject_CallObject(function,
  arguments);
```

I love to do such stuff!

Contact me:
franz@qnipp.com