

# Run interpreters in parallel

Language Summit 2021



Dong-hee Na



Victor Stinner



Red Hat

# Use Cases

# Embed Python

---



- vim
- Blender
- LibreOffice
- pybind11 (C++ applications)

**LINE**

Red Hat

# Subinterpreters



- mod\_wsgi: handle HTTP requests
- weechat plugins (IRC client written in C)

# (A) Embed Python



- Valgrind lists memory leaks **at exit**
- `Py_Finalize()` **must release all memory** allocations done by Python.
- More important for `Py_EndInterpreter()`
- <https://bugs.python.org/issue1635741>  
(created in 2007! work started in 2019)

# (A) Plugin in Python



- Use subinterpreters for plugins
- IRC client **written in C**
- Plugin A uses Python
- Plugin B also uses Python
- Plugins are not aware of each others
- **Unloading** a plugin must **release all memory**

# (B) Run in parallel

---



- Run multiple interpreters in parallel
- One interpreter per thread per CPU
- $N$  CPUs →  $N$  interpreters
- multiprocessing use cases
- Distribute Machine Learning

# (B) Single process



- A single process is more convenient and can be efficient (specific use cases)
- Admin tools are more convenient with 1 process than with N processes
- Some APIs don't work cross-processes
- Windows: creating a thread is faster than creating a process
- macOS: slow multiprocessing spawn

# (B) No shared object



- Problem: concurrent access on object refcnt
- Lock or atomic operation?  
→ performance bottleneck
- Pressure on the same CPU cachelines for common objects: None, True, 1, ()
- Solution: don't share any object, even immutable

# Subinterp drawbacks

---



- On a **crash** (segfault), all interpreters are killed.
- All imported extensions must **support subinterpreters**.

# PEPs

---



- PEP 384: Defining a **Stable ABI**
- PEP 489: **Multi-phase** extension module initialization
- Draft PEP 554: **Multiple Interpreters** in the Stdlib
- PEP 573: **Module State** Access from C Extension Methods
- PEP 630: **Isolating** Extension Modules
- PEP 3121: Extension Module **Initialization** and Finalization

# CAPI & extensions

# Steps



- (A) Convert static types to heap types
- (B) Add module state
- (C) Use PEP 489 multiphase initialization API

# (A) Heap types

---



- PEP 384 & PEP 573: Support HeapType
- PyType\_FromSpec()
- PyType\_FromModuleAndSpec()

# (A) FromSpec

---



Modules defining heap types with PEP  
384 PyType\_FromSpec():

- Python 3.8: 3 modules
- Python 3.9: 9 modules
- Python 3.10: 6 modules

→ Allocate one type per interpreter.

# (A) FromModuleAndSpec



Modules defining heap types with PEP  
573 PyType\_FromModuleAndSpec():

- Python 3.8: 0 modules
- Python 3.9: 2 modules
- Python 3.10: 32 modules

→ Get the module in a method getting  
the type or an instance.

# (A) Static types

---



Modules still defining types as static types:

- Python 3.8: 48 modules
- Python 3.9: 39 modules
- Python 3.10: only 16 modules (!)

# (B) Module state



- Modules using PEP 573 module state:
- Python 3.8: 8 modules
- Python 3.9: 23 modules
- Python 3.10: 48 modules (!)

→ Multiple instances of a single extension module can be created (ex: one per interpreter).

# (C) Multiphase init



Modules using the PEP 489 multiphase initialization API:

- Python 3.8: 3 modules
- Python 3.9: 30 modules
- Python 3.10: 72 modules

\_abc extension

# \_abc module example



- Convert static type into heap type
- Define module state
- Convert module to use multi-phase initialization

# \_abc heap type



- PyType\_Slot and PyType\_Spec must be implemented to define heap types.
- To manage heap type memory:
  - Py\_tp\_new
  - Py\_tp\_dealloc
  - Py\_tp\_traverse (!)
  - Py\_tp\_clearmust be implemented carefully

# PyType\_Slot



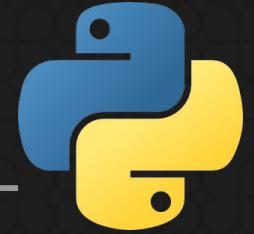
```
static PyType_Slot  
_abc_data_type_spec_slots[] = {  
{Py_tp_doc, (void *)abc_data_doc},  
{Py_tp_new, abc_data_new},  
{Py_tp_dealloc, abc_data_dealloc},  
{Py_tp_traverse, abc_data_traverse},  
{Py_tp_clear, abc_data_clear},  
{0, 0}  
};
```

# PyType\_Spec



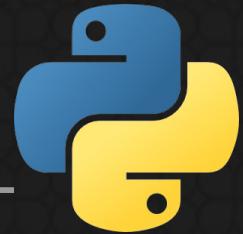
```
static PyType_Spec _abc_data_type_spec = {  
    .name = "__abc.__abc_data",  
    .basicsize = sizeof(__abc_data),  
    .flags = Py_TPFLAGS_DEFAULT  
        | Py_TPFLAGS_HAVE_GC,  
    .slots = __abc_data_type_spec_slots,  
};
```

# \_abc module state



- Module state should have heap type attribute.
- Also, get\_abc\_state() API should be implemented, since accessing module state is frequently needed.

# \_abc module state



```
typedef struct {
    PyTypeObject *_abc_data_type;
    unsigned long abc_invalidation_counter;
} _abcmodule_state;

static inline _abcmodule_state*
get_abc_state(PyObject *module)
{
    void *state = PyModule_GetState(module);
    assert(state != NULL);
    return (_abcmodule_state *)state;
}
```



# \_abc PyModuleDef



- Define `_abcmodule` to implement PEP 489 **multiphase initialization**
- To manage heap type
  - `m_traverse`
  - `m_clear`
  - `m_free`should be implemented carefully

# \_abc PyModuleDef



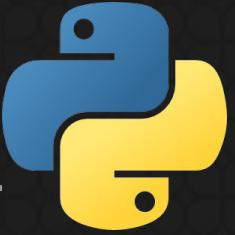
```
static struct PyModuleDef _abcmodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "_abc",
    .m_doc = _abc__doc__,
    .m_size = sizeof(_abcmodule_state),
    .m_methods = _abcmodule_methods,
    .m_slots = _abcmodule_slots,
    .m_traverse = _abcmodule_traverse,
    .m_clear = _abcmodule_clear,
    .m_free = _abcmodule_free,
};
```



# \_abc exec func



- Initialize heap type through Py\_mod\_exec API
- If initialization is failed return -1 if not return 0



# \_abc exec func

```
static int _abcmodule_exec(PyObject *module) {
    _abcmodule_state *state;
    state = get_abc_state(module);
    state->abc_invalidation_counter = 0;
    state->_abc_data_type =
        PyType_FromModuleAndSpec(module,
        &_abc_data_type_spec, NULL);
    if (state->_abc_data_type == NULL) {
        return -1;
    }
    return 0;
}
```



# Get module state



Python 3.8:

```
static unsigned long abc_invalidation_counter;

static PyObject*
_abc_get_cache_token_impl(PyObject *module)
{
    return PyLong_FromUnsignedLong(
        abc_invalidation_counter);
}
```



# Get module state



Python 3.9:

```
static PyObject*
_abc_get_cache_token_impl(PyObject *module)
{
    _abcmodule_state *state;
    state = get_abc_state(module);
    return PyLong_FromUnsignedLong(
        state->abc_invalidation_counter);
}
```

# Get module state

---



- Python 3.8: 50.2 ns
- Python 3.10: 50.8 ns (+0.6 ns)

```
import pyperf
runner = pyperf.Runner()
runner.timeit(
    name='bench _abc',
    setup = 'import _abc',
    stmt=_abc.get_cache_token()')
```

Work done

# Per-interp free lists

---



- float
- tuple, list, dict, slice
- frame, context, asynchronous generator
- MemoryError

# Per-interp singletons

---



- small **integer** ([-5; 256])
- empty **bytes** string
- empty **Unicode** string
- empty **tuple**
- single **byte** char (b'\x00' – b'\xFF')
- single **Unicode** char (U+0000 – U+00FF)

# Per-interpreter ...

---



- slice cache
- pending calls
- type attribute lookup cache
- interned strings:  
`PyUnicode_InternInPlace()`
- identifiers: `_PyUnicode_FromId()`

# Per-interpreter state

---



- ast
- gc
- parser
- warnings

# Proof of Concept

---



- Same factorial function on 4 CPUs
- Sequential: 1.99 sec +- 0.01 sec (ref)
- Threads: 3.15 sec +- 0.97 sec (1.5x slower)
- Multiprocessing: 560 ms +- 12 ms (3.6x faster)
- Subinterpreters: 583 ms +- 7 ms (3.4x faster)
- Subinterpreters weren't optimized at all

# TODO

# TODO (easy)



- Convert remaining **extensions** and **static types**
- Make `_PyArg_Parser` per-interpreter
- **GIL** itself (“already done”)
- Fix unknown bugs ;-)
- Easy but lot of small issues to fix

# TODO (hard)



- Remove **static types** from the public C API
- Make `None`, `True`, `False` **singletons** per interpreter
- Get the Python Thread State (`tstate`) from a thread local storage (`TLS`)

# Public static types

---



- Remove static types from the public C API
- Replace `&PyLong_Type` with `PyLong_GetType()`
- Guido's idea: use `&PyHeapType.ht_type` for `&PyLong_Type`
- Need a PEP if the C API is broken.
- <https://bugs.python.org/issue40601>

# None singleton



- Add an **if** to Py\_INCREF/Py\_DECREF  
→ 10% slower & CPU cacheline pressure
- `#define Py_None Py_GetNone()`  
→ no API issue!
- `Py_GetNone() { return interp->none; }`
- <https://bugs.python.org/issue39511>  
Draft PR 18301

# Get tstate from TLS



- `_PyThreadState_GET()` perf issue
- C11 `_Thread_local` and `<threads.h>`  
**thread\_local**
- x86: single MOV with FS register
- Fallback: `pthread_getspecific()`
- Function call for extensions
- <https://bugs.python.org/issue40522>  
Draft PR 23976



# Open questions



- Need a PEP for the overall isolated interpreters **design** and **rationale**.
- Extensions wrapping C libraries with **shared states**: need a **lock** somewhere.
- Another Python 2 vs Python 3 mess? No.
- Opt-in feature, **adoption can be slow**.

# Future

# Later

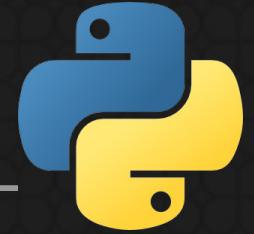
---



- API to share Python objects
- Share data and use one Python object per interpreter with locks
- Support spawning subprocesses (fork)

# Questions?

---



Play with:

```
./configure  
  --with-experimental-isolated-subinterpreters  
#ifdef EXPERIMENTAL_ISOLATED_SUBINTERPRETERS
```

**LINE**



# Bonus

# Performance

---



- Compare Python 3.8, 3.9 and 3.10 at [speed.python.org](https://speed.python.org) (macro benchmarks).
- Benchmarks and microbenchmarks were run on individual changes: no significant overhead.

# Fix daemon threads

---



- Random crashes at Python exit when using daemon threads
- `take_gil()` now checks in 3 places if the current thread must exit immediately (if Python exited)
- Don't read any Python internal structure after Python exited (freed memory)

# Get empty tuple



```
static PyObject* tuple_get_empty(void)
{
    PyInterpreterState *interp;
    interp = _PyInterpreterState_GET();
    PyObject *op = interp->tuple.free_list[0];
    return Py_NewRef(op);
}
```