// Licensed to the Apache Software Foundation (ASF) under one

// or more contributor license agreements. See the NOTICE file

// distributed with this work for additional information

// regarding copyright ownership. The ASF licenses this file

// to you under the Apache License, Version 2.0 (the

// "License"); you may not use this file except in compliance

// with the License. You may obtain a copy of the License at

//

// http://www.apache.org/licenses/LICENSE-2.0

//

// Unless required by applicable law or agreed to in writing,

// software distributed under the License is distributed on an

// "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY

// KIND, either express or implied. See the License for the

// specific language governing permissions and limitations

// under the License.

#include "io.h"

#include <cstdint>

#include <cstdlib>

#include <memory>

#include <mutex>

#include <string>

#include "arrow/io/memory.h"

#include "arrow/memory\_pool.h"

#include "arrow/status.h"

#include "arrow/util/logging.h"

#include "arrow/python/common.h"

#include "arrow/python/pyarrow.h"

namespace arrow {

using arrow::io::TransformInputStream;

namespace py {

// ----------------------------------------------------------------------

// Python file

// A common interface to a Python file-like object. Must acquire GIL before

// calling any methods

class PythonFile {

public:

explicit PythonFile(PyObject\* file) : file\_(file), checked\_read\_buffer\_(false) {

Py\_INCREF(file);

}

Status CheckClosed() const {

if (!file\_) {

return Status::Invalid("operation on closed Python file");

}

return Status::OK();

}

Status Close() {

if (file\_) {

PyObject\* result = cpp\_PyObject\_CallMethod(file\_.obj(), "close", "()");

Py\_XDECREF(result);

file\_.reset();

PY\_RETURN\_IF\_ERROR(StatusCode::IOError);

}

return Status::OK();

}

Status Abort() {

file\_.reset();

return Status::OK();

}

bool closed() const {

if (!file\_) {

return true;

}

PyObject\* result = PyObject\_GetAttrString(file\_.obj(), "closed");

if (result == NULL) {

// Can't propagate the error, so write it out and return an arbitrary value

PyErr\_WriteUnraisable(NULL);

return true;

}

int ret = PyObject\_IsTrue(result);

Py\_XDECREF(result);

if (ret < 0) {

PyErr\_WriteUnraisable(NULL);

return true;

}

return ret != 0;

}

Status Seek(int64\_t position, int whence) {

RETURN\_NOT\_OK(CheckClosed());

// NOTE: `long long` is at least 64 bits in the C standard, the cast below is

// therefore safe.

// whence: 0 for relative to start of file, 2 for end of file

PyObject\* result = cpp\_PyObject\_CallMethod(file\_.obj(), "seek", "(Li)",

static\_cast<long long>(position), whence);

Py\_XDECREF(result);

PY\_RETURN\_IF\_ERROR(StatusCode::IOError);

return Status::OK();

}

Status Read(int64\_t nbytes, PyObject\*\* out) {

RETURN\_NOT\_OK(CheckClosed());

PyObject\* result = cpp\_PyObject\_CallMethod(file\_.obj(), "read", "(L)",

static\_cast<long long>(nbytes));

PY\_RETURN\_IF\_ERROR(StatusCode::IOError);

\*out = result;

return Status::OK();

}

Status ReadBuffer(int64\_t nbytes, PyObject\*\* out) {

PyObject\* result = cpp\_PyObject\_CallMethod(file\_.obj(), "read\_buffer", "(L)",

static\_cast<long long>(nbytes));

PY\_RETURN\_IF\_ERROR(StatusCode::IOError);

\*out = result;

return Status::OK();

}

Status Write(const void\* data, int64\_t nbytes) {

RETURN\_NOT\_OK(CheckClosed());

// Since the data isn't owned, we have to make a copy

PyObject\* py\_data =

PyBytes\_FromStringAndSize(reinterpret\_cast<const char\*>(data), nbytes);

PY\_RETURN\_IF\_ERROR(StatusCode::IOError);

PyObject\* result = cpp\_PyObject\_CallMethod(file\_.obj(), "write", "(O)", py\_data);

Py\_XDECREF(py\_data);

Py\_XDECREF(result);

PY\_RETURN\_IF\_ERROR(StatusCode::IOError);

return Status::OK();

}

Status Write(const std::shared\_ptr<Buffer>& buffer) {

RETURN\_NOT\_OK(CheckClosed());

PyObject\* py\_data = wrap\_buffer(buffer);

PY\_RETURN\_IF\_ERROR(StatusCode::IOError);

PyObject\* result = cpp\_PyObject\_CallMethod(file\_.obj(), "write", "(O)", py\_data);

Py\_XDECREF(py\_data);

Py\_XDECREF(result);

PY\_RETURN\_IF\_ERROR(StatusCode::IOError);

return Status::OK();

}

Result<int64\_t> Tell() {

RETURN\_NOT\_OK(CheckClosed());

PyObject\* result = cpp\_PyObject\_CallMethod(file\_.obj(), "tell", "()");

PY\_RETURN\_IF\_ERROR(StatusCode::IOError);

int64\_t position = PyLong\_AsLongLong(result);

Py\_DECREF(result);

// PyLong\_AsLongLong can raise OverflowError

PY\_RETURN\_IF\_ERROR(StatusCode::IOError);

return position;

}

std::mutex& lock() { return lock\_; }

bool HasReadBuffer() {

if (!checked\_read\_buffer\_) { // we don't want to check this each time

has\_read\_buffer\_ = PyObject\_HasAttrString(file\_.obj(), "read\_buffer") == 1;

checked\_read\_buffer\_ = true;

}

return has\_read\_buffer\_;

}

private:

std::mutex lock\_;

OwnedRefNoGIL file\_;

bool has\_read\_buffer\_;

bool checked\_read\_buffer\_;

};

// ----------------------------------------------------------------------

// Seekable input stream

PyReadableFile::PyReadableFile(PyObject\* file) { file\_.reset(new PythonFile(file)); }

// The destructor does not close the underlying Python file object, as

// there may be multiple references to it. Instead let the Python

// destructor do its job.

PyReadableFile::~PyReadableFile() {}

Status PyReadableFile::Abort() {

return SafeCallIntoPython([this]() { return file\_->Abort(); });

}

Status PyReadableFile::Close() {

return SafeCallIntoPython([this]() { return file\_->Close(); });

}

bool PyReadableFile::closed() const {

bool res;

Status st = SafeCallIntoPython([this, &res]() {

res = file\_->closed();

return Status::OK();

});

return res;

}

Status PyReadableFile::Seek(int64\_t position) {

return SafeCallIntoPython([=] { return file\_->Seek(position, 0); });

}

Result<int64\_t> PyReadableFile::Tell() const {

return SafeCallIntoPython([=]() -> Result<int64\_t> { return file\_->Tell(); });

}

Result<int64\_t> PyReadableFile::Read(int64\_t nbytes, void\* out) {

return SafeCallIntoPython([=]() -> Result<int64\_t> {

OwnedRef bytes;

RETURN\_NOT\_OK(file\_->Read(nbytes, bytes.ref()));

PyObject\* bytes\_obj = bytes.obj();

DCHECK(bytes\_obj != NULL);

Py\_buffer py\_buf;

if (!PyObject\_GetBuffer(bytes\_obj, &py\_buf, PyBUF\_ANY\_CONTIGUOUS)) {

const uint8\_t\* data = reinterpret\_cast<const uint8\_t\*>(py\_buf.buf);

std::memcpy(out, data, py\_buf.len);

int64\_t len = py\_buf.len;

PyBuffer\_Release(&py\_buf);

return len;

} else {

return Status::TypeError(

"Python file read() should have returned a bytes object or an object "

"supporting the buffer protocol, got '",

Py\_TYPE(bytes\_obj)->tp\_name, "' (did you open the file in binary mode?)");

}

});

}

Result<std::shared\_ptr<Buffer>> PyReadableFile::Read(int64\_t nbytes) {

return SafeCallIntoPython([=]() -> Result<std::shared\_ptr<Buffer>> {

OwnedRef buffer\_obj;

if (file\_->HasReadBuffer()) {

RETURN\_NOT\_OK(file\_->ReadBuffer(nbytes, buffer\_obj.ref()));

} else {

RETURN\_NOT\_OK(file\_->Read(nbytes, buffer\_obj.ref()));

}

DCHECK(buffer\_obj.obj() != NULL);

return PyBuffer::FromPyObject(buffer\_obj.obj());

});

}

Result<int64\_t> PyReadableFile::ReadAt(int64\_t position, int64\_t nbytes, void\* out) {

std::lock\_guard<std::mutex> guard(file\_->lock());

return SafeCallIntoPython([=]() -> Result<int64\_t> {

RETURN\_NOT\_OK(Seek(position));

return Read(nbytes, out);

});

}

Result<std::shared\_ptr<Buffer>> PyReadableFile::ReadAt(int64\_t position, int64\_t nbytes) {

std::lock\_guard<std::mutex> guard(file\_->lock());

return SafeCallIntoPython([=]() -> Result<std::shared\_ptr<Buffer>> {

RETURN\_NOT\_OK(Seek(position));

return Read(nbytes);

});

}

Result<int64\_t> PyReadableFile::GetSize() {

return SafeCallIntoPython([=]() -> Result<int64\_t> {

ARROW\_ASSIGN\_OR\_RAISE(int64\_t current\_position, file\_->Tell());

RETURN\_NOT\_OK(file\_->Seek(0, 2));

ARROW\_ASSIGN\_OR\_RAISE(int64\_t file\_size, file\_->Tell());

// Restore previous file position

RETURN\_NOT\_OK(file\_->Seek(current\_position, 0));

return file\_size;

});

}

// ----------------------------------------------------------------------

// Output stream

PyOutputStream::PyOutputStream(PyObject\* file) : position\_(0) {

file\_.reset(new PythonFile(file));

}

// The destructor does not close the underlying Python file object, as

// there may be multiple references to it. Instead let the Python

// destructor do its job.

PyOutputStream::~PyOutputStream() {}

Status PyOutputStream::Abort() {

return SafeCallIntoPython([=]() { return file\_->Abort(); });

}

Status PyOutputStream::Close() {

return SafeCallIntoPython([=]() { return file\_->Close(); });

}

bool PyOutputStream::closed() const {

bool res;

Status st = SafeCallIntoPython([this, &res]() {

res = file\_->closed();

return Status::OK();

});

return res;

}

Result<int64\_t> PyOutputStream::Tell() const { return position\_; }

Status PyOutputStream::Write(const void\* data, int64\_t nbytes) {

return SafeCallIntoPython([=]() {

position\_ += nbytes;

return file\_->Write(data, nbytes);

});

}

Status PyOutputStream::Write(const std::shared\_ptr<Buffer>& buffer) {

return SafeCallIntoPython([=]() {

position\_ += buffer->size();

return file\_->Write(buffer);

});

}

// ----------------------------------------------------------------------

// Foreign buffer

Status PyForeignBuffer::Make(const uint8\_t\* data, int64\_t size, PyObject\* base,

std::shared\_ptr<Buffer>\* out) {

PyForeignBuffer\* buf = new PyForeignBuffer(data, size, base);

if (buf == NULL) {

return Status::OutOfMemory("could not allocate foreign buffer object");

} else {

\*out = std::shared\_ptr<Buffer>(buf);

return Status::OK();

}

}

// ----------------------------------------------------------------------

// TransformInputStream::TransformFunc wrapper

struct TransformFunctionWrapper {

TransformFunctionWrapper(TransformCallback cb, PyObject\* arg)

: cb\_(std::move(cb)), arg\_(std::make\_shared<OwnedRefNoGIL>(arg)) {

Py\_INCREF(arg);

}

Result<std::shared\_ptr<Buffer>> operator()(const std::shared\_ptr<Buffer>& src) {

return SafeCallIntoPython([=]() -> Result<std::shared\_ptr<Buffer>> {

std::shared\_ptr<Buffer> dest;

cb\_(arg\_->obj(), src, &dest);

RETURN\_NOT\_OK(CheckPyError());

return dest;

});

}

protected:

// Need to wrap OwnedRefNoGIL because std::function needs the callable

// to be copy-constructible...

TransformCallback cb\_;

std::shared\_ptr<OwnedRefNoGIL> arg\_;

};

std::shared\_ptr<::arrow::io::InputStream> MakeTransformInputStream(

std::shared\_ptr<::arrow::io::InputStream> wrapped, TransformInputStreamVTable vtable,

PyObject\* handler) {

TransformInputStream::TransformFunc transform(

TransformFunctionWrapper{std::move(vtable.transform), handler});

return std::make\_shared<TransformInputStream>(std::move(wrapped), std::move(transform));

}

std::shared\_ptr<StreamWrapFunc> MakeStreamTransformFunc(TransformInputStreamVTable vtable,

PyObject\* handler) {

TransformInputStream::TransformFunc transform(

TransformFunctionWrapper{std::move(vtable.transform), handler});

StreamWrapFunc func = [transform](std::shared\_ptr<::arrow::io::InputStream> wrapped) {

return std::make\_shared<TransformInputStream>(wrapped, transform);

};

return std::make\_shared<StreamWrapFunc>(func);

}

} // namespace py

} // namespace arrow