Shared Memory Based JITLink Memory Manager

Student: Anubhab Ghosh

Mentors: Vassil Vassilev, Lang Hames, Stefan Gränitz
JITLink is a Just-In-Time linker.
- It takes multiple object code units and links them together.
- It constructs the result directly in memory.
- The resulting code is usually immediately run.

It uses a LinkGraph as memory representation.
- It consists of nodes like Addressable, Block, Symbol.
- Relocations are represented by edges.
- Sections consist of symbols and blocks.

It can work with two different processes.
- An executor process that is running the resultant code.
- A controller process that performs the linking and controls the executor.
- The communication happens through Executor Process Control, an RPC scheme.
Memory management

- Memory allocation is performed using the JITLinkMemoryManager interface.
- It has 3 steps
  - Allocate
    - Reserves address space
  - Finalize
    - Copies link result from working memory to executor
    - Runs initialisation actions
  - Deallocate
    - Runs deinitialization actions
    - Deallocate memory
- Initialization and deinitialization actions are just functions that will be executed in the context of the target process.

```cpp
class JITLinkMemoryManager {
public:
  class FinalizedAlloc {
    orc::ExecutorAddr release();
  };

class InFlightAlloc {
  virtual void abandon(OnAbandonedFunction OnAbandoned) = 0;
  virtual void finalize(OnFinalizedFunction OnFinalized) = 0;
}

virtual void allocate(const JITLinkDylib *JD, LinkGraph &G,
                       OnAllocatedFunction OnAllocated) = 0;

virtual void deallocate(std::vector<FinalizedAlloc> Allocs,
                         OnDeallocatedFunction OnDeallocated) = 0;
};
```

When multiple processes are involved, this is implemented with the **EPCGenericJITLinkMemoryManager** and **SimpleExecutorMemoryManager**.
The Executor Process side

- Implemented using a bootstrap service.
- 3 primary functions: allocate, finalize and deallocate
- Deallocation actions are also transferred during finalization.
The Controller Process

The controller process side is implemented in EPCGenericJITLinkMemoryManager.

It mainly consists of RPC calls to the methods of SimpleExecutorMemoryManager.

```cpp
void finalize(OnFinalizedFunction OnFinalize) override {
    tptypes::FinalizeRequest FR;
    for (auto &KV : Segs) {
        assert(KV.second.contentSize <= std::numeric_limits<size_t>::max());
        FR.Segments.push_back(tptypes::SegFinalizeRequest{
            tptypes::toWireProtectionFlags(
                tsysMemoryProtectionFlags(KV.first.getMemProt())),
            KV.second.Addr,
            alignTo(KV.second.contentSize + KV.second.ZeroFillSize,
                    Parent.EPC.getPageSize()),
            {KV.second.WorkingMem, static_cast<size_t>(KV.second.contentSize)) });
    }
    // Transfer allocation actions.
    std::swap(FR.Actions, G.allocActions());

    Parent.EPC.callSpsWrapperAsync<
        SPSSimpleExecutorMemoryManagerFinalizeSignature<
            Parent.SAs.Finalize,
            [OnFinalize = std::move(OnFinalize), AllocAddr = this->AllocAddr](
                Error SerializationErr, Error FinalizeErr) mutable {
                //FIXME: Release abandoned alloc.
                if (SerializationErr) {
                    cantFail(std::move(FinalizeErr));
                    OnFinalize(std::move(SerializationErr));
                } else if (FinalizeErr) {
                    OnFinalize(std::move(FinalizeErr));
                } else {
                    OnFinalize(FinalizedAlloc(AllocAddr));
                },
            Parent.SAs.Allocator, std::move(FR))};
```
```c
int FromExecutor[2];

pid_t ChildPID;

// Create pipes to/from the executor...
if (!pipeToExecutor || pipeFromExecutor != 0)
    return make_error_string_error("Unable to create pipe for executor",
        inconvertibleErrorCode());

ChildPID = fork();

if (ChildPID == 0) {
    // In the child...
    close(pipeToExecutor);
    close(pipeFromExecutor);
    // Execute the child process.
    std::unique_ptr<char[]> ExecutorPath, FDSpecifier;
    { ExecutorPath = std::make_unique<char[]>(OOfProcessExecutor.size() + 1);
        strcpy(ExecutorPath.get(), OOfProcessExecutor.data());
        std::string FDSpecifierStr("filenames=");
        FDSpecifierStr += std::to_string(ExecutorPath.size());
        FDSpecifierStr += ",
        FDSpecifierStr += std::to_string(pipeToExecutor.size());
        char* const Args[] = {ExecutorPath.get(), FDSpecifier.get(), nullptr};
        int RC = execvp(ExecutorPath.get(), Args);
    }

    addrinfo Hints[0];
    Hints.ai_family = AF_INET;
    Hints.ai_socktype = SOCK_STREAM;
    Hints.ai_flags = AI_PASSIVE;
    if (getaddrinfo(nullptr, PortStr.c_str(), &Hints, &Al)) {
        std::error_code EC = std::error_code(EC)
            << " Error setting up bind address: "
            << std::strerror(Errno) << "\n";
        exit(1);
    }
    // Create a socket from first addinfo structure returned by getaddrinfo.
    int SockFD;
    if ((SockFD = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        std::error_code EC = std::error_code(Errno)
            << " Error creating socket: "
            << std::strerror(Errno) << "\n";
        exit(1);
    }
    // Avoid "Address already in use" errors.
    const int Yes = 1;
    if (setsockopt(SockFD, SOL_SOCKET, SO_REUSEADDR, &Yes, sizeof(Yes)) != 0) {
        std::error_code EC = std::error_code(Errno)
            << " Error calling setsockopt: "
            << std::strerror(Errno) << "\n";
        exit(1);
    }
    // Bind the socket to the desired port.
    if (bind(SockFD, AI_to_addr, AI_to_addr.len) < 0) {
        std::error_code EC = std::error_code(Errno)
            << " Error on binding: "
            << std::strerror(Errno) << "\n";
        exit(1);
    }
    // Listen for incoming connections.
    static const expr int ConnectionQueueLen = 1;
    listen(SockFD, ConnectionQueueLen);
```
The plan

- A MemoryMapper interface with implementations based on
  - Shared memory
    - When both executor and controller process share same physical memory
  - Regular memory allocation APIs
    - When the resultant code is executed in the same process
    - Useful for unit tests
  - EPC
    - Required when the executor and controller process run with different physical memory
    - Resultant code is transferred to the executor process over the EPC channel

- A JITLinkMemoryManager implementation that can use any MemoryMapper
  - It will allocate large chunks of memory using MemoryMapper and divide into smaller chunks
  - Better support for small code model by keeping everything close in memory
MemoryMapper Interface

- **Reserve**
  - Reserves executor address space
  - Creates shared memory or regular allocation
- **Prepare**
  - Provides pointer to working memory for use by the linker
  - Could be already mapped shared memory or just regular temporary memory to be copied
- **Initialize**
  - Transfers memory contents to executor and runs initialization actions
  - No-op for in-process or shared memory
- **Deinitialize**
  - Runs deinitialization actions
- **Release**
  - Release executor address space

```cpp
namespace llvm {
namespace err {

class MemoryMapper {
public:
  struct AllocInfo {
    struct SegInfo {
      ExecutorAddr Diff Offset;
      const char *WorkingMem;
      size_t ContentSize;
      size_t ZeroFillSize;
      unsigned Prot;
    };
    ExecutorAddr MappingBase;
    std::vector<SegInfo> Segments;
    shared::AllocActions Actions;
  };

  using OnReservedFunction = unique_function<void(Expected<ExecutorAddrRange>>);
  virtual void reserve(size_t NumBytes, OnReservedFunction OnReserved) = 0;

  virtual char *prepare(ExecutorAddr Addr, size_t ContentSize) = 0;

  using OnInitializedFunction = unique_function<void(Expected<ExecutorAddr>>);
  virtual void initialize(AllocInfo &AI,
                          OnInitializedFunction OnInitialized) = 0;

  using OnDeinitializedFunction = unique_function<void(Error>>;
  virtual void deinitialize(std::vector<ExecutorAddr> &Allocations,
                            OnDeinitializedFunction OnDeinitialized) = 0;

  using OnReleasedFunction = unique_function<void(Error>>;
  virtual void release(std::vector<ExecutorAddr> &Reservations,
                       OnReleasedFunction OnRelease) = 0;
};
```
Current Progress

- MemoryMapper interface in review
- InProcessMemoryMapper implementation using sys::Memory APIs in review
- SharedMemoryMapper needs to be adapted to new MemoryMapper interface design (Currently working)
Thank you