# Shared Memory Based JITLink Memory Manager

Student: Anubhab Ghosh

Mentors: Vassil Vassilev, Lang Hames, Stefan Gränitz

# JITLink

- 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 Adressable, Block, Symbol.
  - Relocations are represented by edges.
  - Sections consists 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
- Intialization and deinitialization actions are just functions that will be executed in the context of the target process.

```
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

```cpp
/// Simple page-based allocator.
class SimpleExecutorMemoryManager : public ExecutorBootstrapService {
public:
  virtual ~SimpleExecutorMemoryManager();

  Expected<ExecutorAddr> allocate(uint64_t Size);
  Error finalize(tpctypes::FinalizeRequest &FR);
  Error deallocate(const std::vector<ExecutorAddr> &Bases);
```

```cpp
struct SegFinalizeRequest {
  WireProtectionFlags Prot;
  ExecutorAddr Addr;
  uint64_t Size;
  ArrayRef<char> Content;
};

struct FinalizeRequest {
  std::vector<SegFinalizeRequest> Segments;
  shared::AllocActions Actions;
};
```

```cpp
Expected<ExecutorAddr> SimpleExecutorMemoryManager::allocate(uint64_t Size) {
  std::error_code EC;
  auto MB = sys::Memory::allocateMappedMemory(
      Size, nullptr, sys::Memory::MF_READ | sys::Memory::MF_WRITE, EC);
  return ExecutorAddr::fromPtr(MB.base());
}
```

```cpp
Error SimpleExecutorMemoryManager::finalize(tpctypes::FinalizeRequest &FR) {
  // Copy content and apply permissions.
  for (auto &Seg : FR.Segments) {
    char *Mem = Seg.Addr.toPtr<char *>();
    memcpy(Mem, Seg.Content.data(), Seg.Content.size());
    memset(Mem + Seg.Content.size(), 0, Seg.Size - Seg.Content.size());
    assert(Seg.Size <= std::numeric_limits<size_t>::max());
    if (auto EC = sys::Memory::protectMappedMemory(
            {Mem, static_cast<size_t>(Seg.Size)},
            tpctypes::fromWireProtectionFlags(Seg.Prot)))
      return BailOut(errorCodeToError(EC));
    if (Seg.Prot & tpctypes::WPF_Exec)
      sys::Memory::InvalidateInstructionCache(Mem, Seg.Size);
  }
```

- 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.

```
void finalize(OnFinalizedFunction OnFinalize) override {
  tpctypes::FinalizeRequest FR;
  for (auto &KV : Segs) {
    assert(KV.second.ContentSize <= std::numeric_limits<size_t>::max());
    FR.Segments.push_back(tpctypes::SegFinalizeRequest{
        tpctypes::toWireProtectionFlags(
            toSysMemoryProtectionFlags(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<
      rt::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));
}
```

# EPC implementation under the hood

```cpp
int FromExecutor[2];

pid_t ChildPID;

// Create pipes to/from the executor..
if (pipe(ToExecutor) != 0 || pipe(FromExecutor) != 0)
  return make_error<StringError>("Unable to create pipe for executor",
                                 inconvertibleErrorCode());

ChildPID = fork();

if (ChildPID == 0) {
  // In the child...

  // Close the parent ends of the pipes
  close(ToExecutor[WriteEnd]);
  close(FromExecutor[ReadEnd]);

  // Execute the child process.
  std::unique_ptr<char[]> ExecutorPath, FDSpecifier;
  {
    ExecutorPath = std::make_unique<char[]>(OutOfProcessExecutor.size() + 1);
    strcpy(ExecutorPath.get(), OutOfProcessExecutor.data());

    std::string FDSpecifierStr("filedescs=");
    FDSpecifierStr += utostr(ToExecutor[ReadEnd]);
    FDSpecifierStr += ',';
    FDSpecifierStr += utostr(FromExecutor[WriteEnd]);
    FDSpecifier = std::make_unique<char[]>(FDSpecifierStr.size() + 1);
    strcpy(FDSpecifier.get(), FDSpecifierStr.c_str());
  }

  char *const Args[] = {ExecutorPath.get(), FDSpecifier.get(), nullptr};
  int RC = execvp(ExecutorPath.get(), Args);
```

```cpp
int openListener(std::string Host, std::string PortStr) {
  addrinfo Hints{};
  Hints.ai_family = AF_INET;
  Hints.ai_socktype = SOCK_STREAM;
  Hints.ai_flags = AI_PASSIVE;

  addrinfo *AI;
  if (int EC = getaddrinfo(nullptr, PortStr.c_str(), &Hints, &AI)) {
    errs() << "Error setting up bind address: " << gai_strerror(EC) << "\n";
    exit(1);
  }

  // Create a socket from first addrinfo structure returned by getaddrinfo.
  int SockFD;
  if ((SockFD = socket(AI->ai_family, AI->ai_socktype, AI->ai_protocol)) < 0) {
    errs() << "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(int)) == -1) {
    errs() << "Error calling setsockopt: " << std::strerror(errno) << "\n";
    exit(1);
  }

  // Bind the socket to the desired port.
  if (bind(SockFD, AI->ai_addr, AI->ai_addrlen) < 0) {
    errs() << "Error on binding: " << std::strerror(errno) << "\n";
    exit(1);
  }

  // Listen for incomming connections.
  static constexpr 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

```
namespace llvm {
namespace orc {

class MemoryMapper {
public:
  struct AllocInfo {
    struct SegInfo {
      ExecutorAddrDiff 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