Optimizing App Startup Time

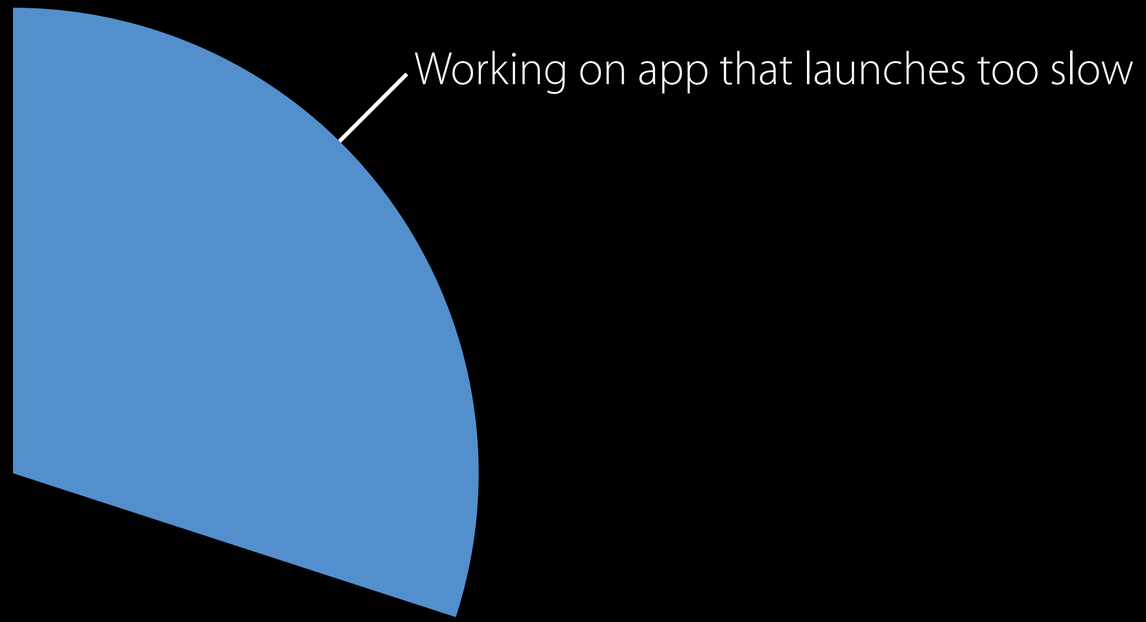
Linkers, loaders, and you

Session 406

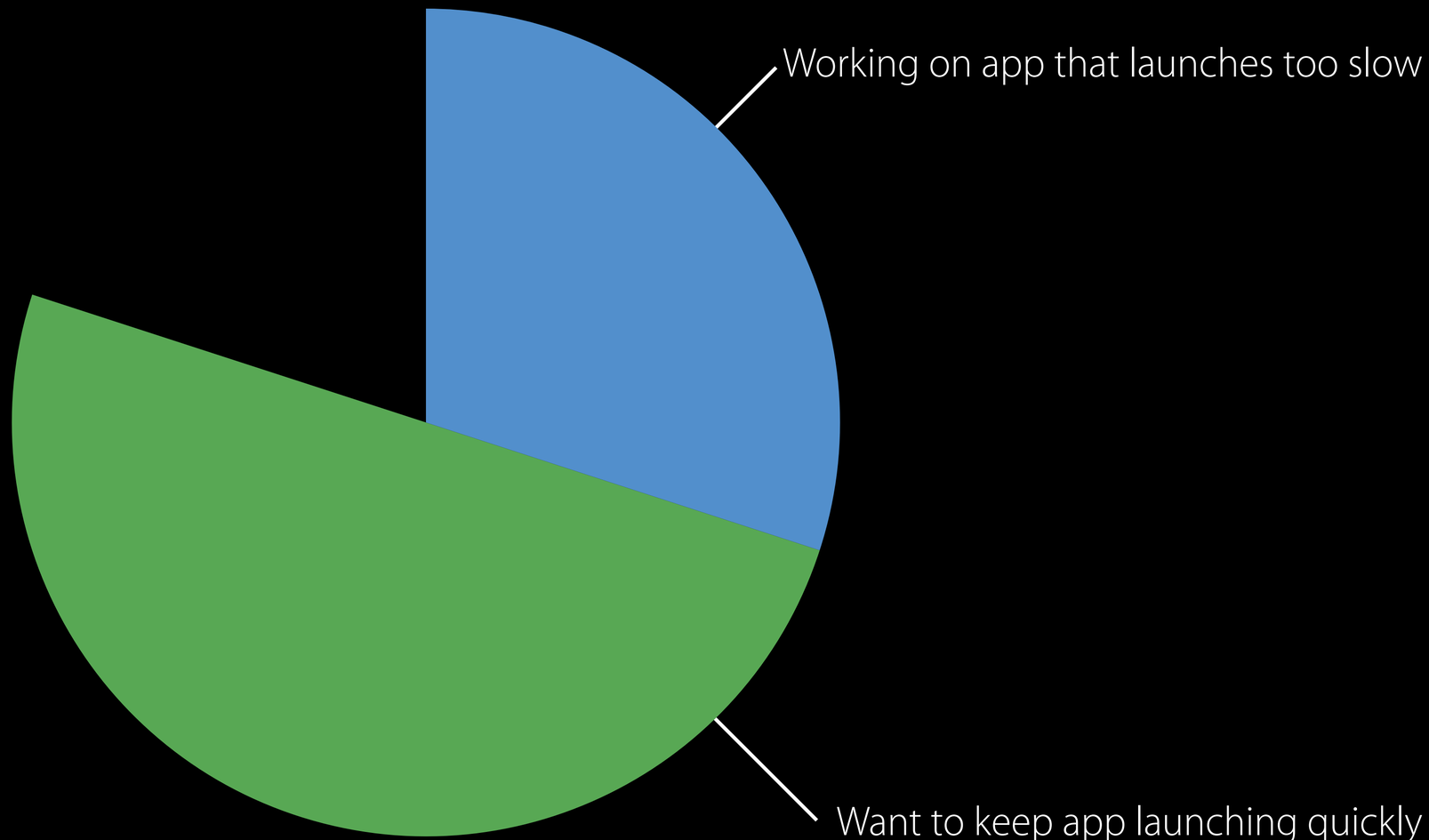
Nick Kledzik Dyld Architect
Louis Gerbarg Dyld Visionary

Audience

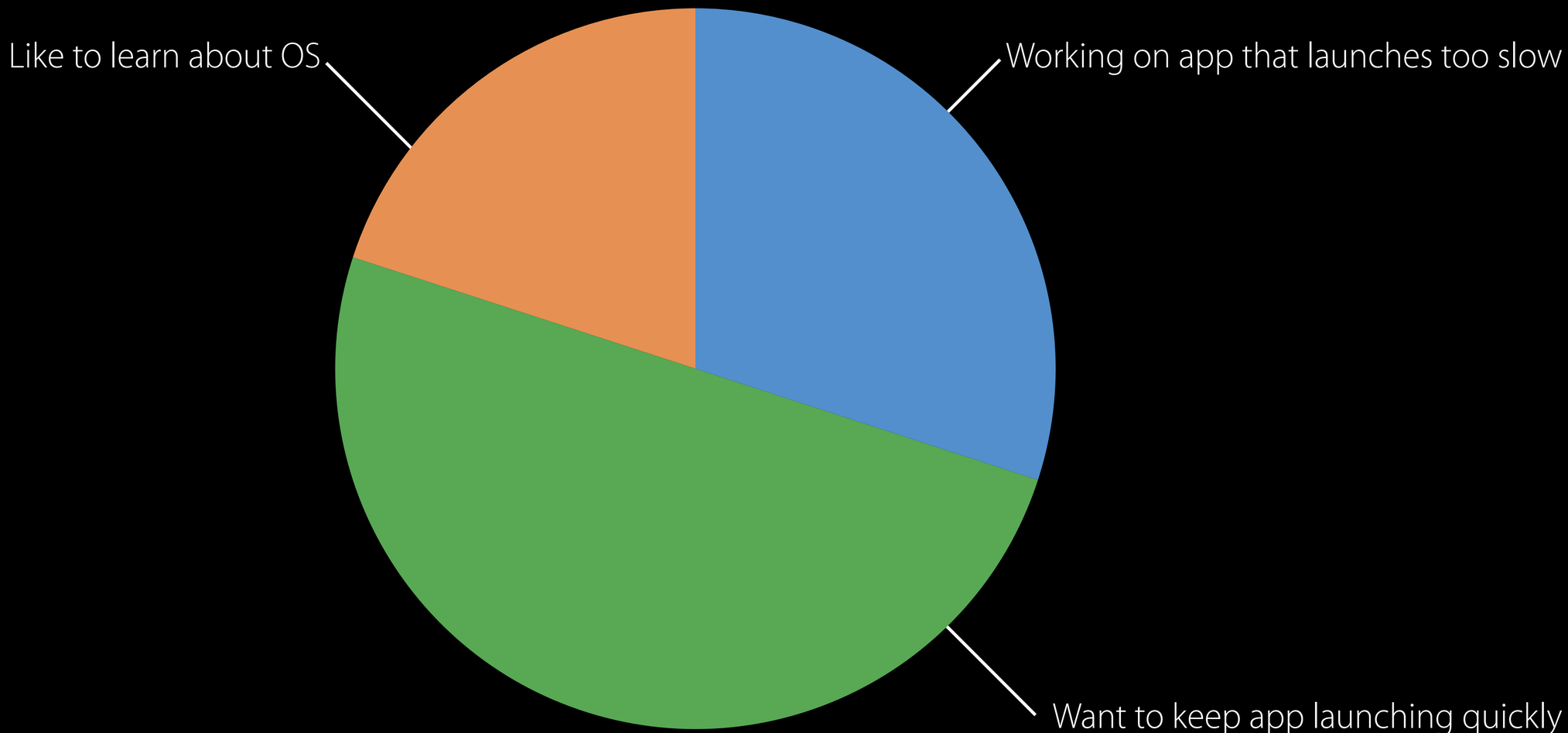
Audience



Audience


Working on app that launches too slow
Want to keep app launching quickly

Audience


Like to learn about OS
Working on app that launches too slow
Want to keep app launching quickly

What You Will Learn

What You Will Learn

Theory

- Everything that happens before main()
- Mach-O format
- Virtual Memory basics
- How Mach-O binaries are loaded and prepared

What You Will Learn

Theory

- Everything that happens before main()
- Mach-O format
- Virtual Memory basics
- How Mach-O binaries are loaded and prepared

Practical

- How to measure
- Optimizing start up time

Crash Course:

Mach-O and Virtual Memory

Mach-O Terminology

File Types:

Mach-O Terminology

File Types:

- **Executable**—Main binary for application

Mach-O Terminology

File Types:

- **Executable**—Main binary for application
- **Dylib**—Dynamic library (aka DSO or DLL)

Mach-O Terminology

File Types:

- **Executable**—Main binary for application
- **Dylib**—Dynamic library (aka DSO or DLL)
- **Bundle**—Dylib that cannot be linked, only `dlopen()`, e.g. plug-ins

Mach-O Terminology

File Types:

- **Executable**—Main binary for application
- **Dylib**—Dynamic library (aka DSO or DLL)
- **Bundle**—Dylib that cannot be linked, only `dlopen()`, e.g. plug-ins

Image—An executable, dylib, or bundle

Mach-O Terminology

File Types:

- **Executable**—Main binary for application
- **Dylib**—Dynamic library (aka DSO or DLL)
- **Bundle**—Dylib that cannot be linked, only `dlopen()`, e.g. plug-ins

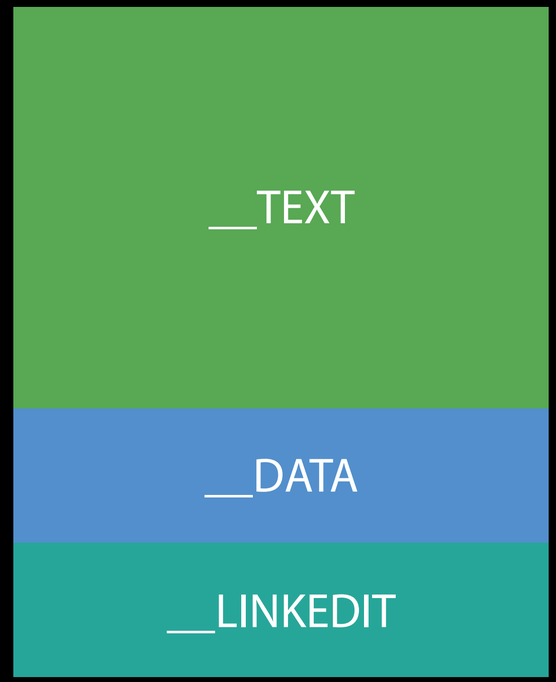
Image—An executable, dylib, or bundle

Framework—Dylib with directory for resources and headers

Mach-O Image File

File divided into segments

- Uppercase names



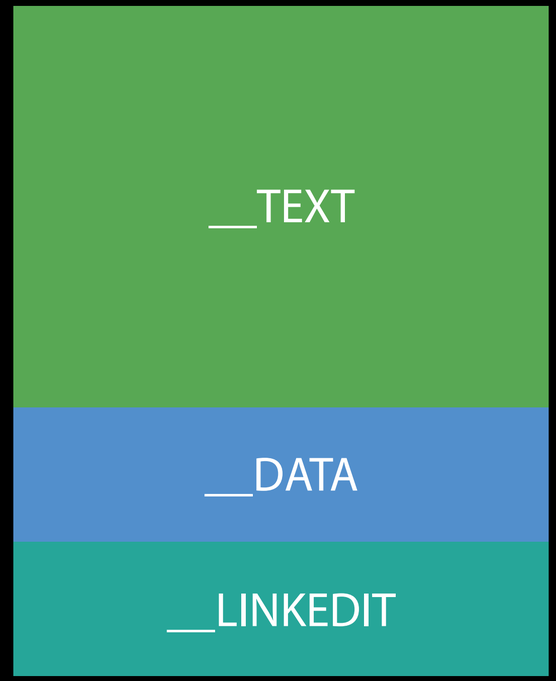
Mach-O Image File

File divided into segments

- Uppercase names

All segments are multiples of page size

- 16KB on arm64
- 4KB elsewhere



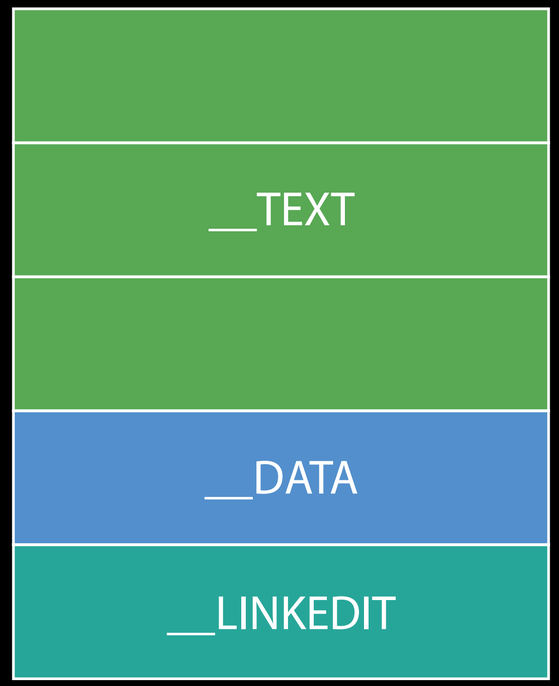
Mach-O Image File

File divided into segments

- Uppercase names

All segments are multiples of page size

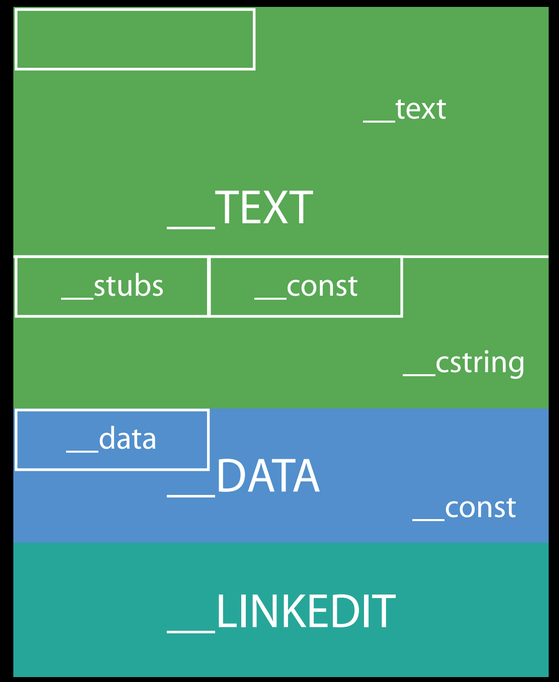
- 16KB on arm64
- 4KB elsewhere



Mach-O Image File

Sections are a subrange of a segment

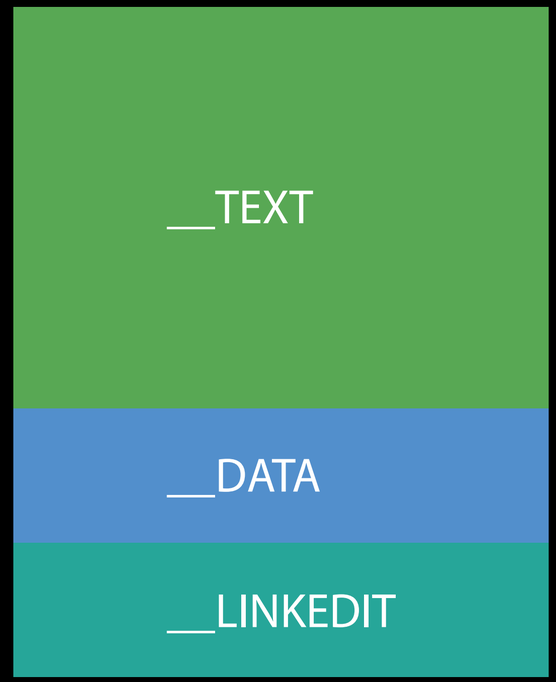
- Lowercase names



Mach-O Image File

Sections are a subrange of a segment

- Lowercase names

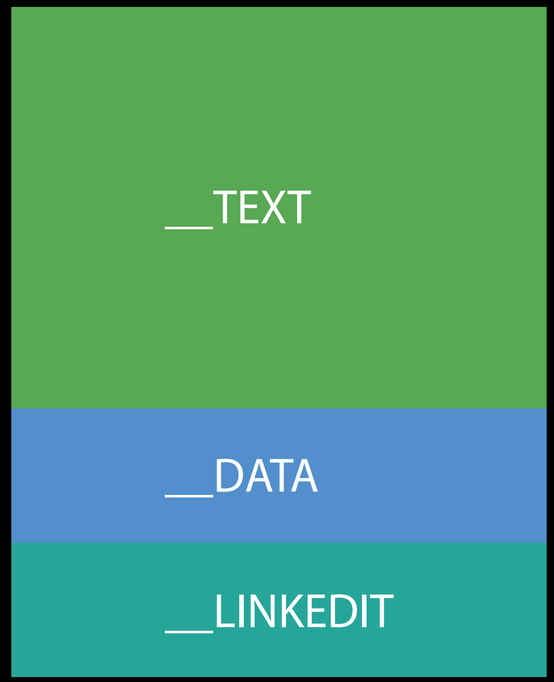


Mach-O Image File

Sections are a subrange of a segment

- Lowercase names

Common segments:



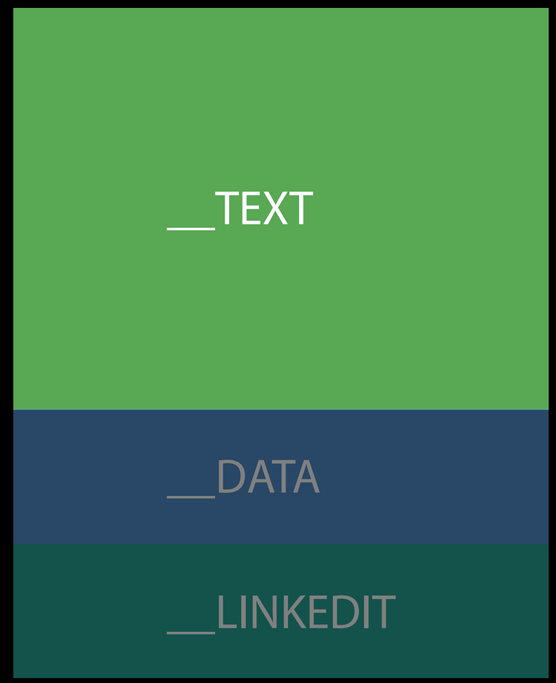
Mach-O Image File

Sections are a subrange of a segment

- Lowercase names

Common segments:

- __TEXT has header, code, and read-only constants



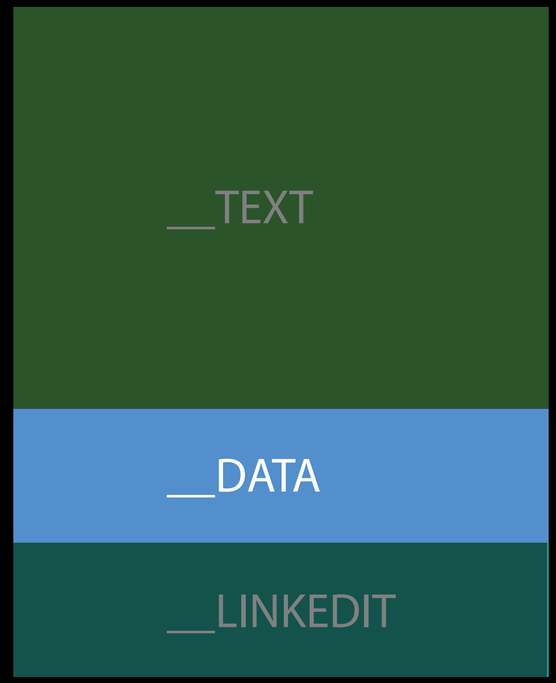
Mach-O Image File

Sections are a subrange of a segment

- Lowercase names

Common segments:

- __TEXT has header, code, and read-only constants
- __DATA has all read-write content: globals, static variables, etc



Mach-O Image File

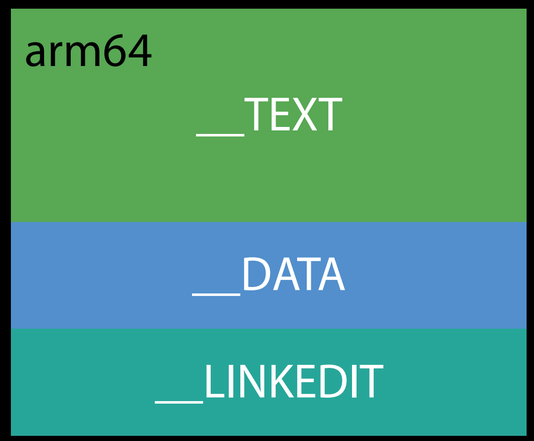
Sections are a subrange of a segment

- Lowercase names

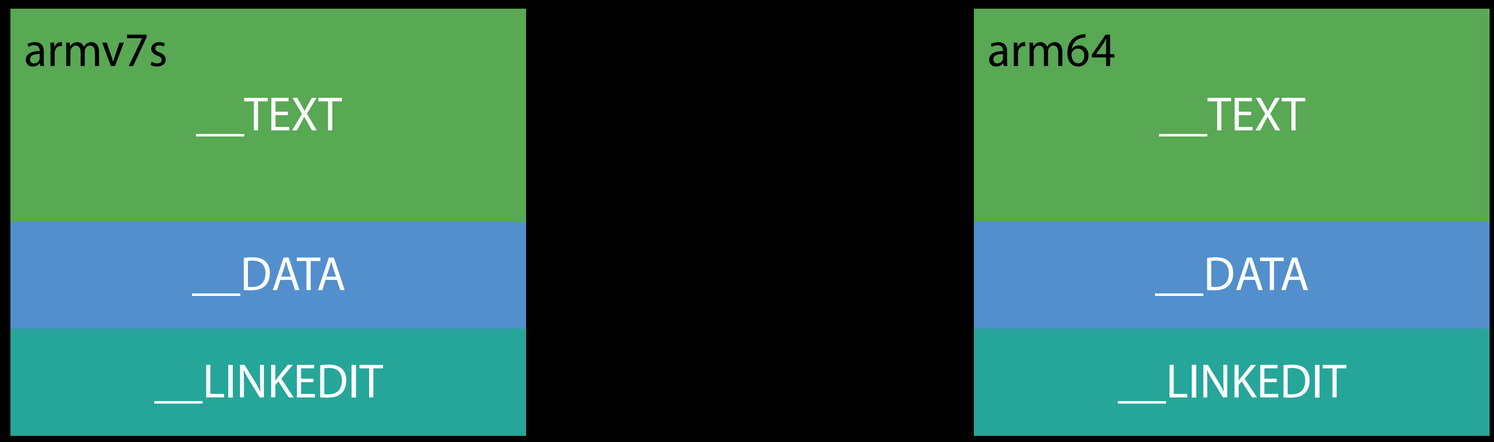
Common segments:

- __TEXT has header, code, and read-only constants
- __DATA has all read-write content: globals, static variables, etc
- __LINKEDIT has "meta data" about how to load the program

Mach-O Universal Files



Mach-O Universal Files

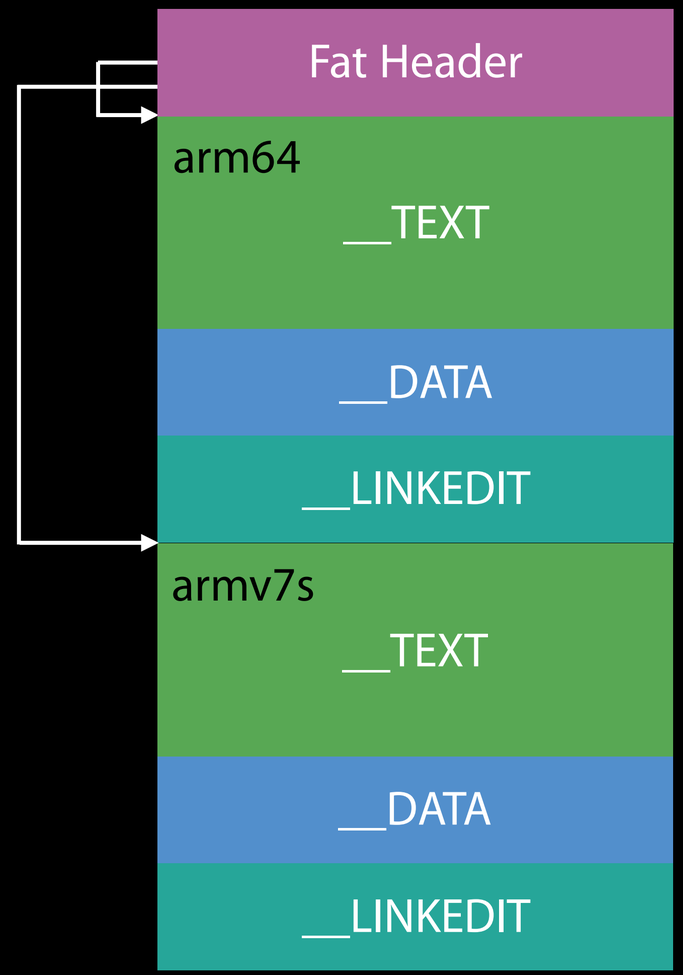


Mach-O Universal Files

Fat Header

- One page in size
- Lists architectures and offsets

Tools and runtimes support fat mach-o files



Virtual Memory

Virtual Memory is a level of indirection

Virtual Memory

Virtual Memory is a level of indirection

Maps per-process addresses to physical RAM (page granularity)

Virtual Memory

Virtual Memory is a level of indirection

Maps per-process addresses to physical RAM (page granularity)

Features:

Virtual Memory

Virtual Memory is a level of indirection

Maps per-process addresses to physical RAM (page granularity)

Features:

- Page fault

Virtual Memory

Virtual Memory is a level of indirection

Maps per-process addresses to physical RAM (page granularity)

Features:

- Page fault
- Same RAM page appears in multiple processes

Virtual Memory

Virtual Memory is a level of indirection

Maps per-process addresses to physical RAM (page granularity)

Features:

- Page fault
- Same RAM page appears in multiple processes
- File backed pages

Virtual Memory

Virtual Memory is a level of indirection

Maps per-process addresses to physical RAM (page granularity)

Features:

- Page fault
- Same RAM page appears in multiple processes
- File backed pages
  - `mmap()`

Virtual Memory

Virtual Memory is a level of indirection

Maps per-process addresses to physical RAM (page granularity)

Features:

- Page fault
- Same RAM page appears in multiple processes
- File backed pages
  - `mmap()`
  - lazy reading

Virtual Memory

Virtual Memory is a level of indirection

Maps per-process addresses to physical RAM (page granularity)

Features:

- Page fault
- Same RAM page appears in multiple processes
- File backed pages
  - `mmap()`
  - lazy reading
- Copy-On-Write (COW)

Virtual Memory

Virtual Memory is a level of indirection

Maps per-process addresses to physical RAM (page granularity)

Features:

- Page fault
- Same RAM page appears in multiple processes
- File backed pages
  - `mmap()`
  - lazy reading
- Copy-On-Write (COW)
- Dirty vs. clean pages

Virtual Memory

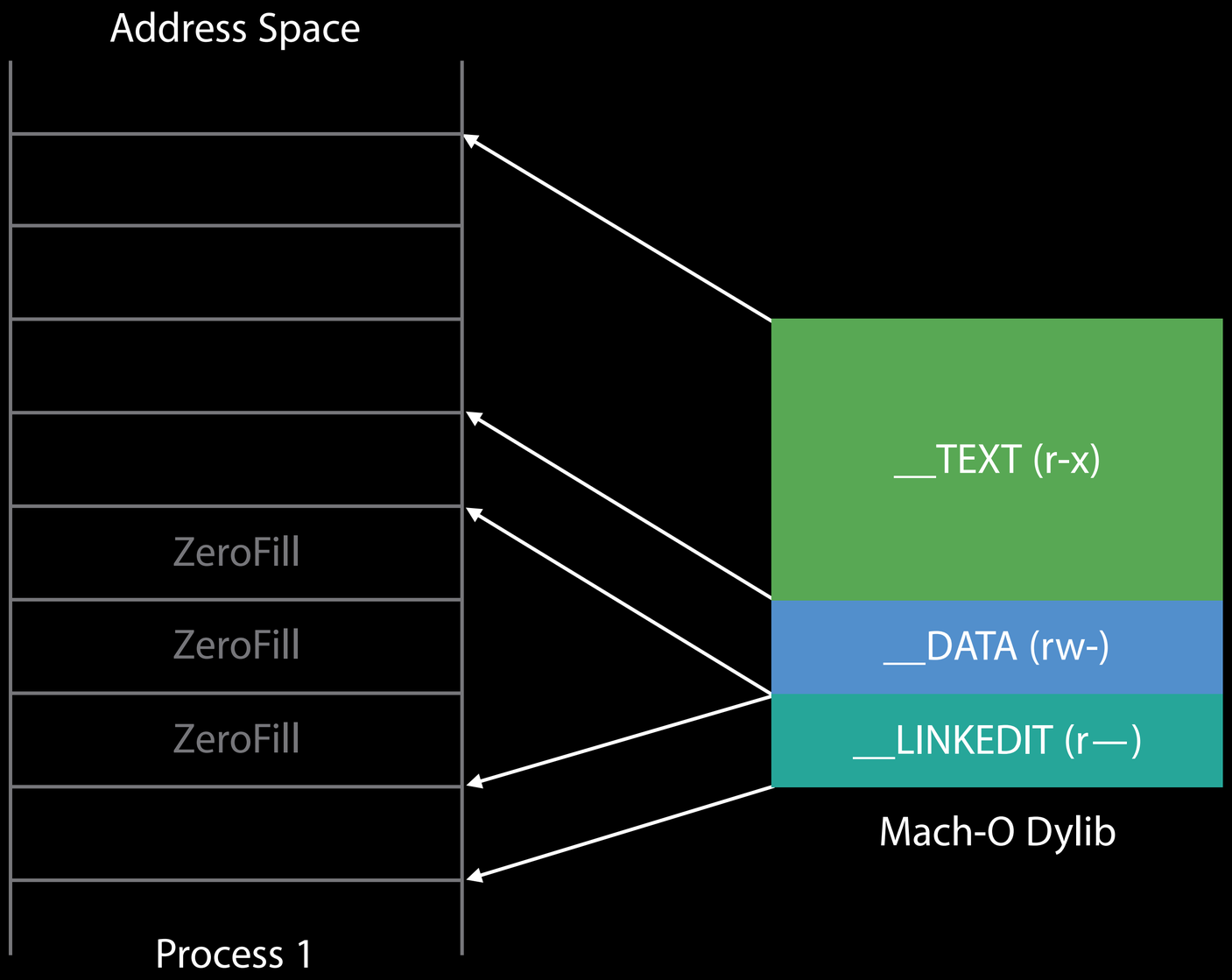
Virtual Memory is a level of indirection

Maps per-process addresses to physical RAM (page granularity)

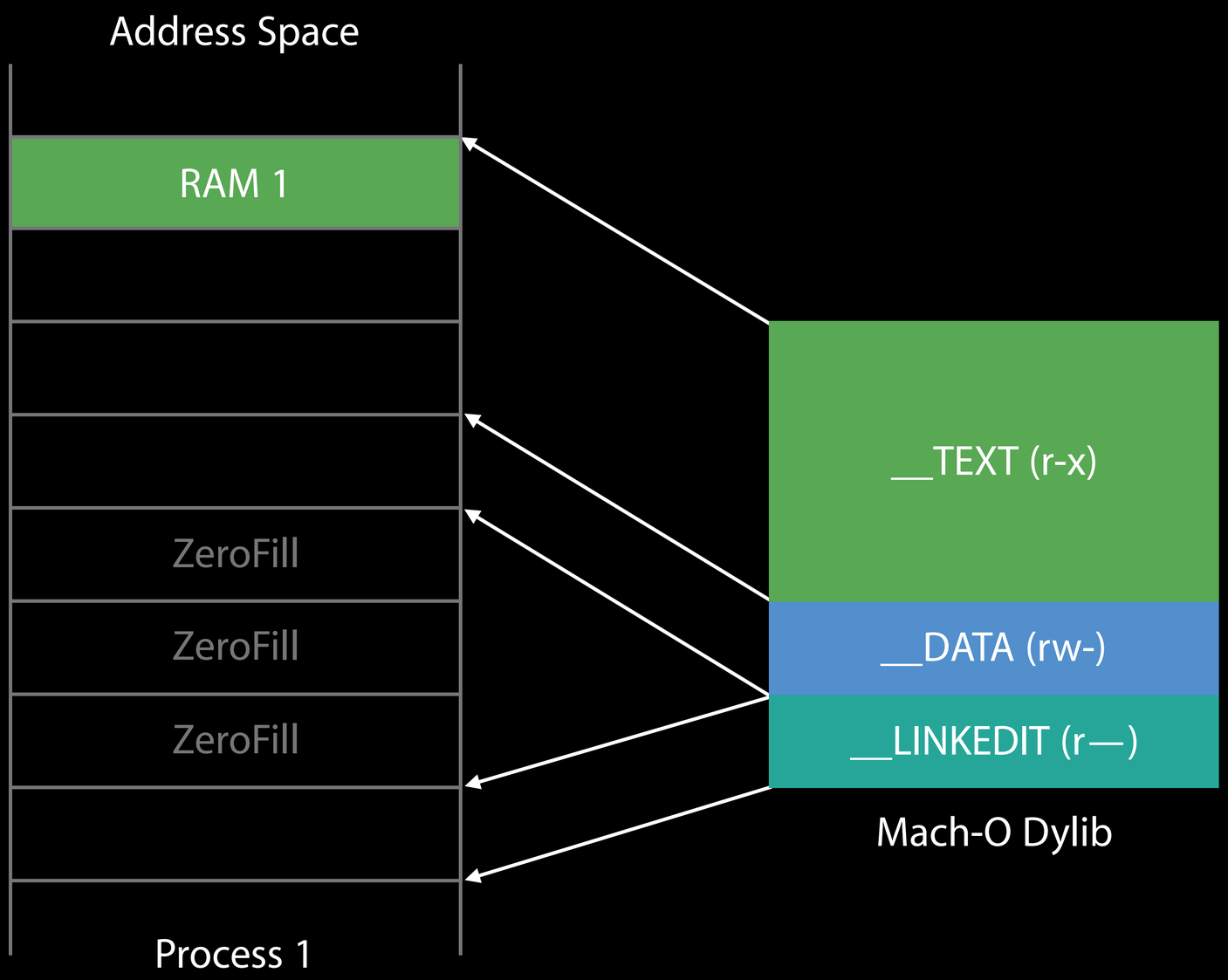
Features:

- Page fault
- Same RAM page appears in multiple processes
- File backed pages
  - `mmap()`
  - lazy reading
- Copy-On-Write (COW)
- Dirty vs. clean pages
- Permissions: rwx

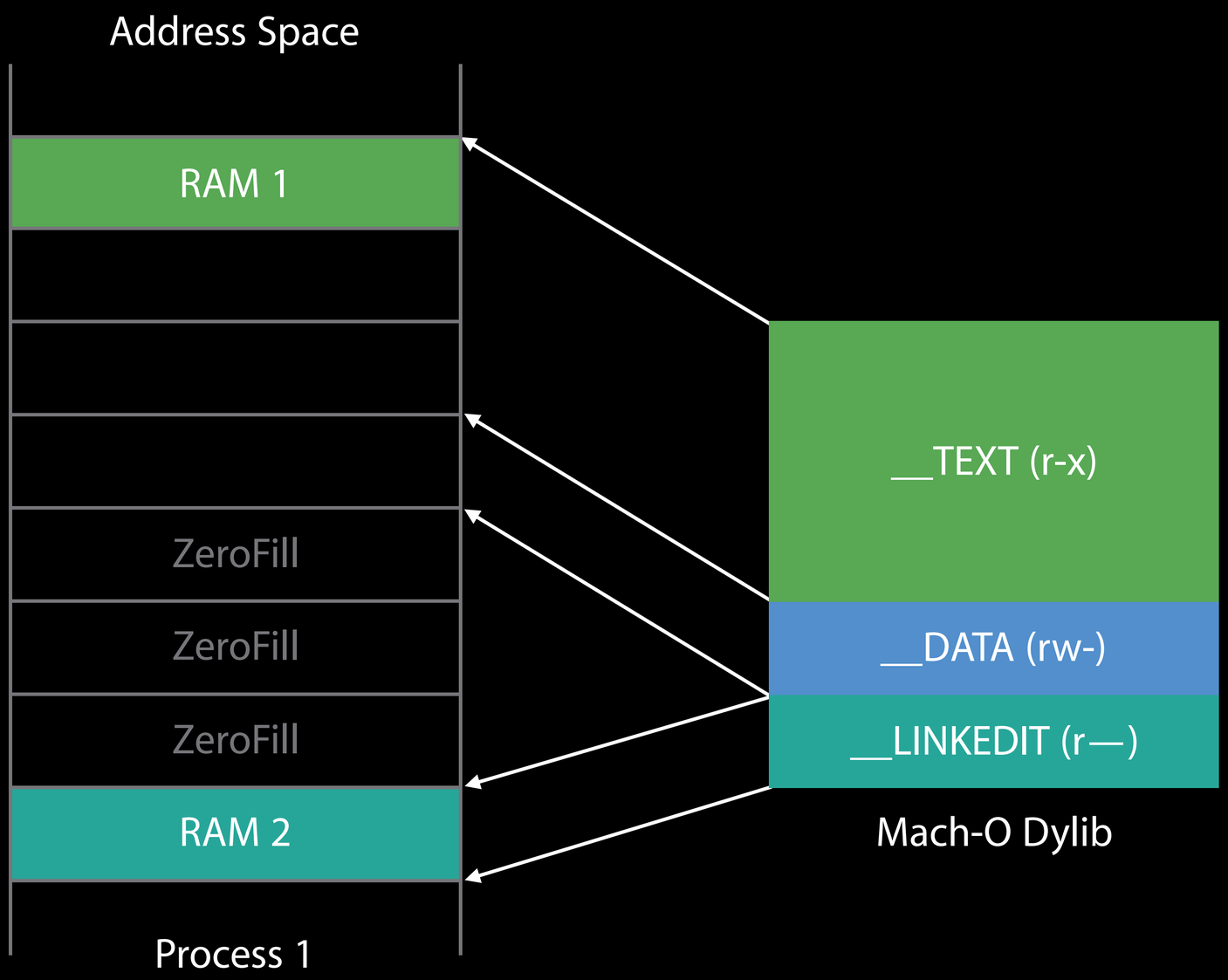
Mach-O Image Loading



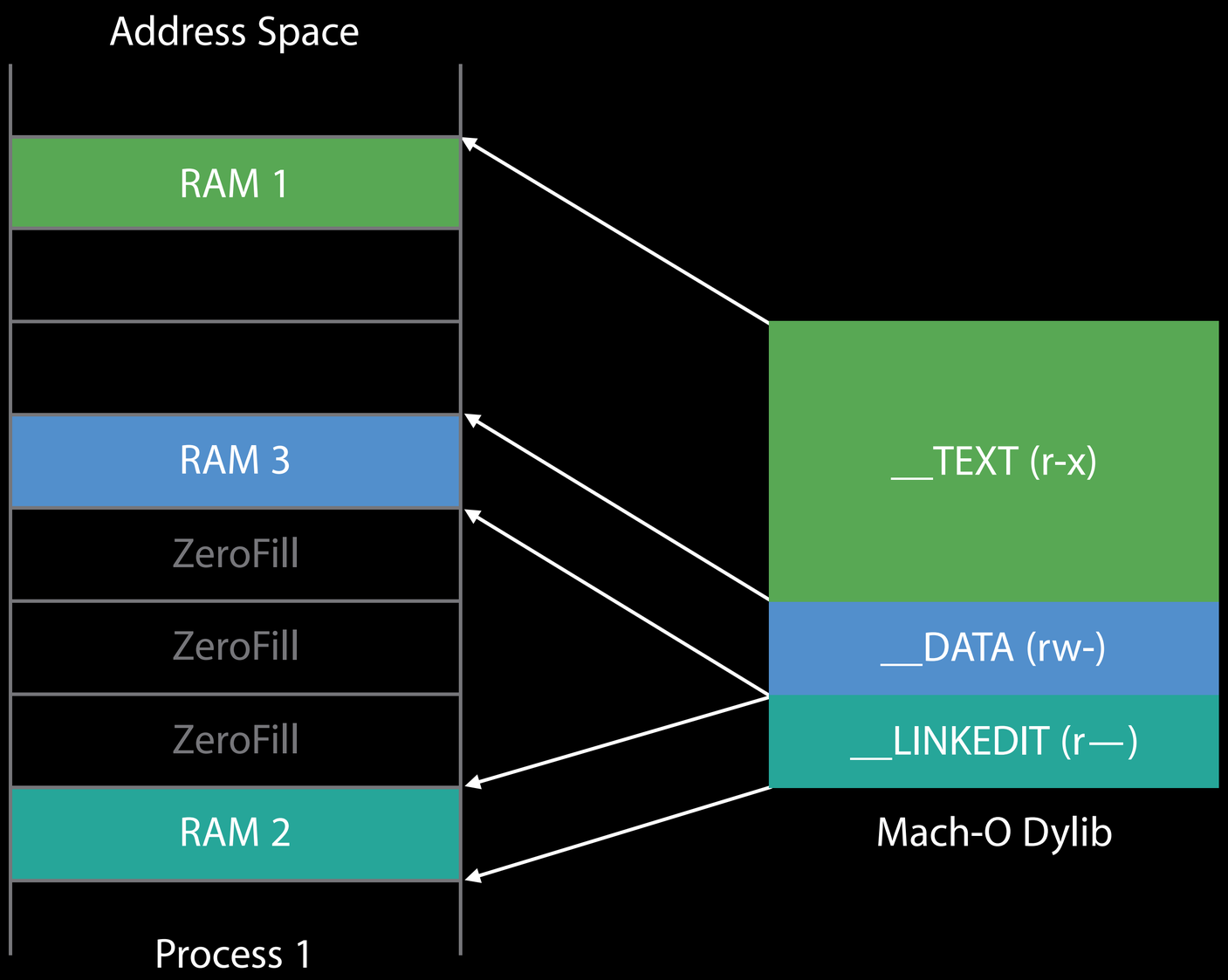
Mach-O Image Loading



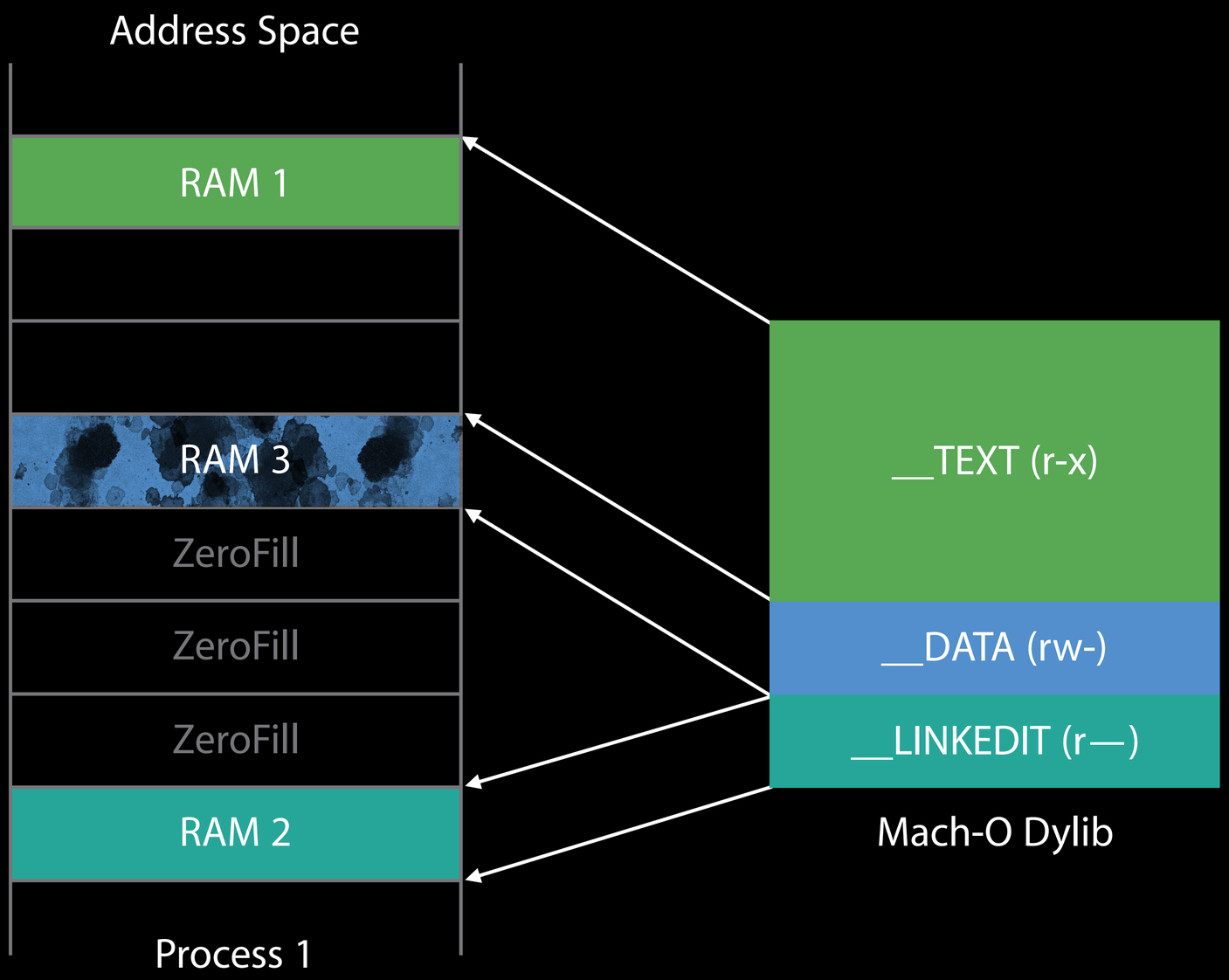
Mach-O Image Loading



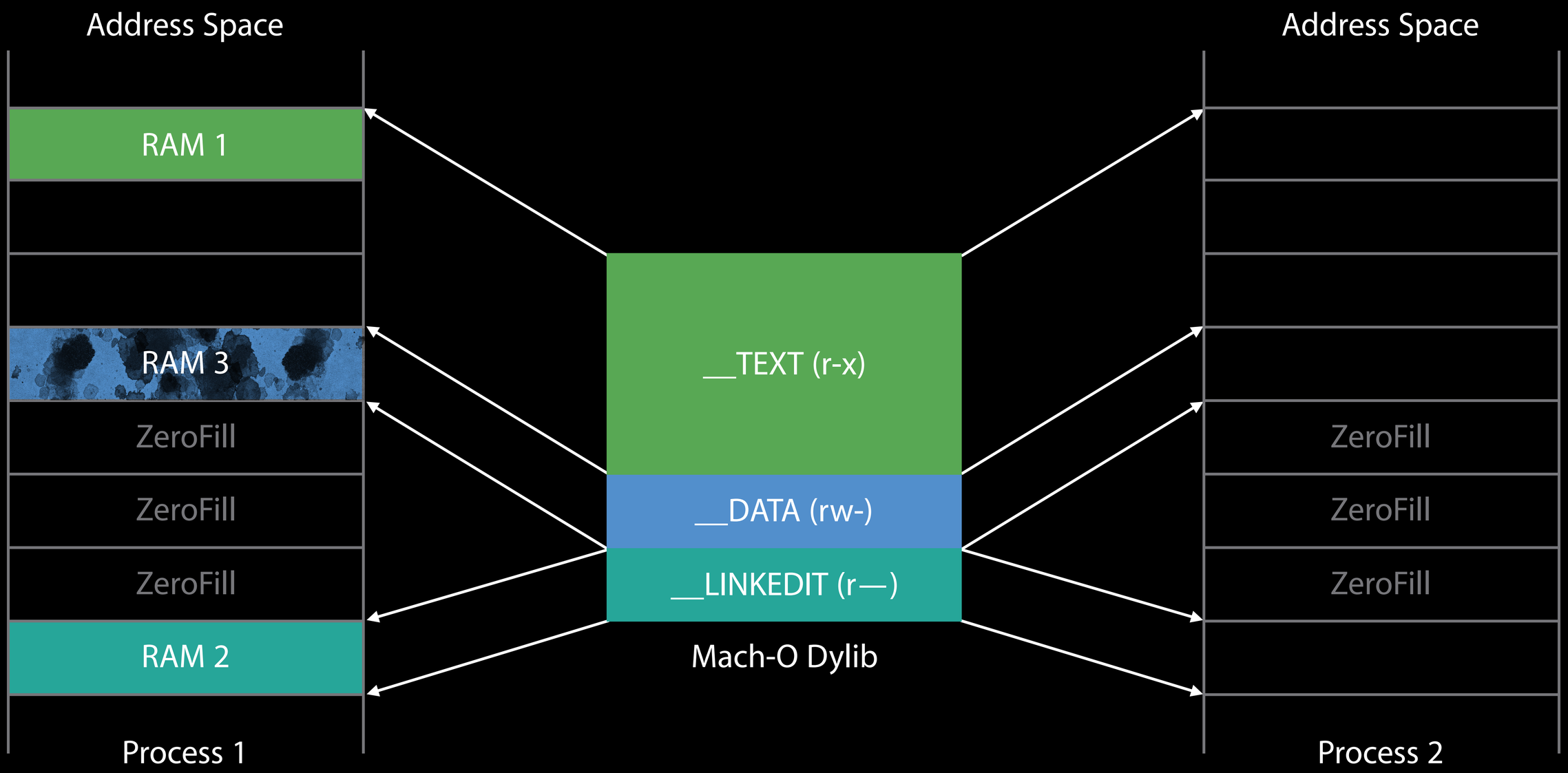
Mach-O Image Loading



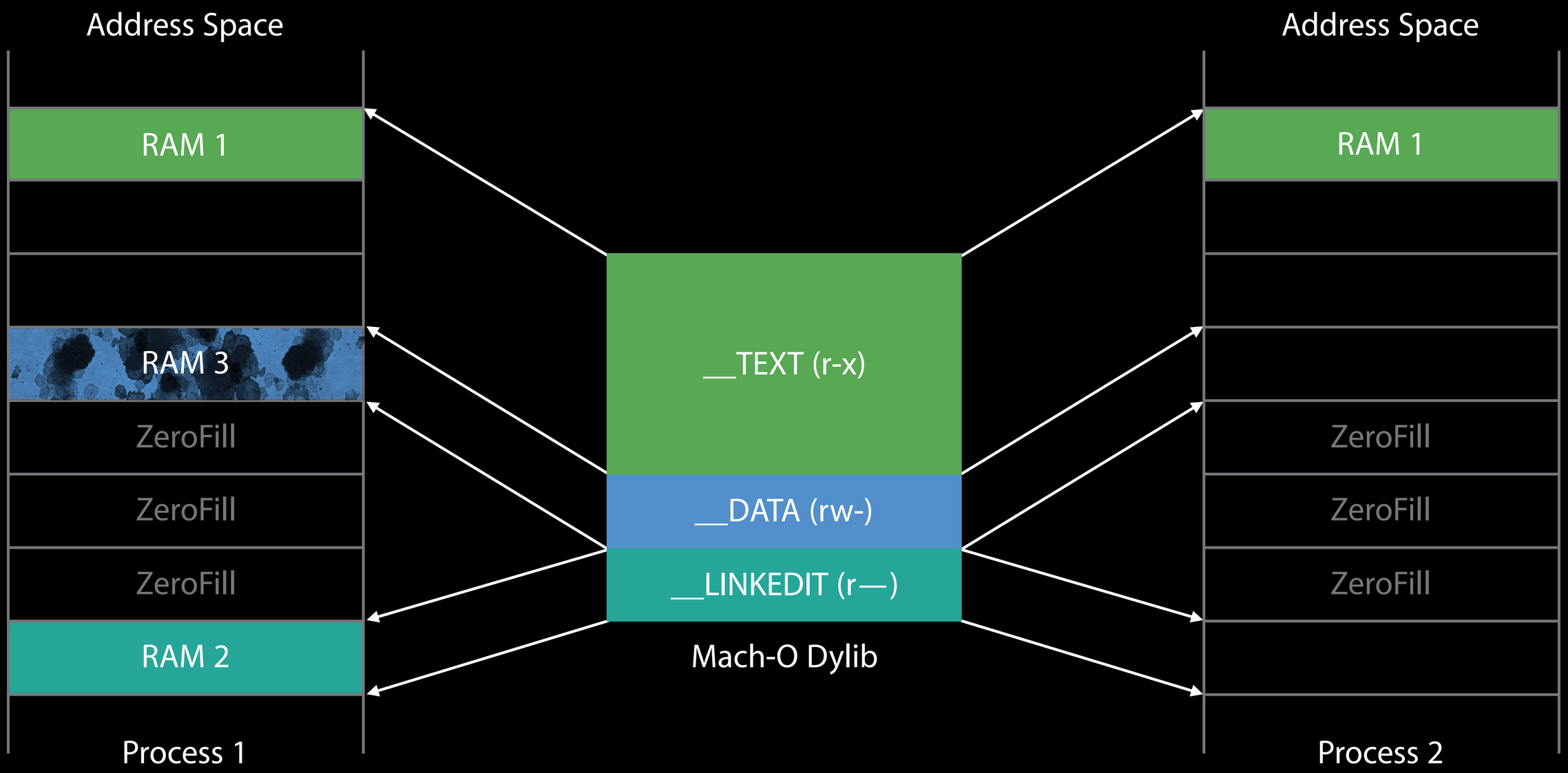
Mach-O Image Loading



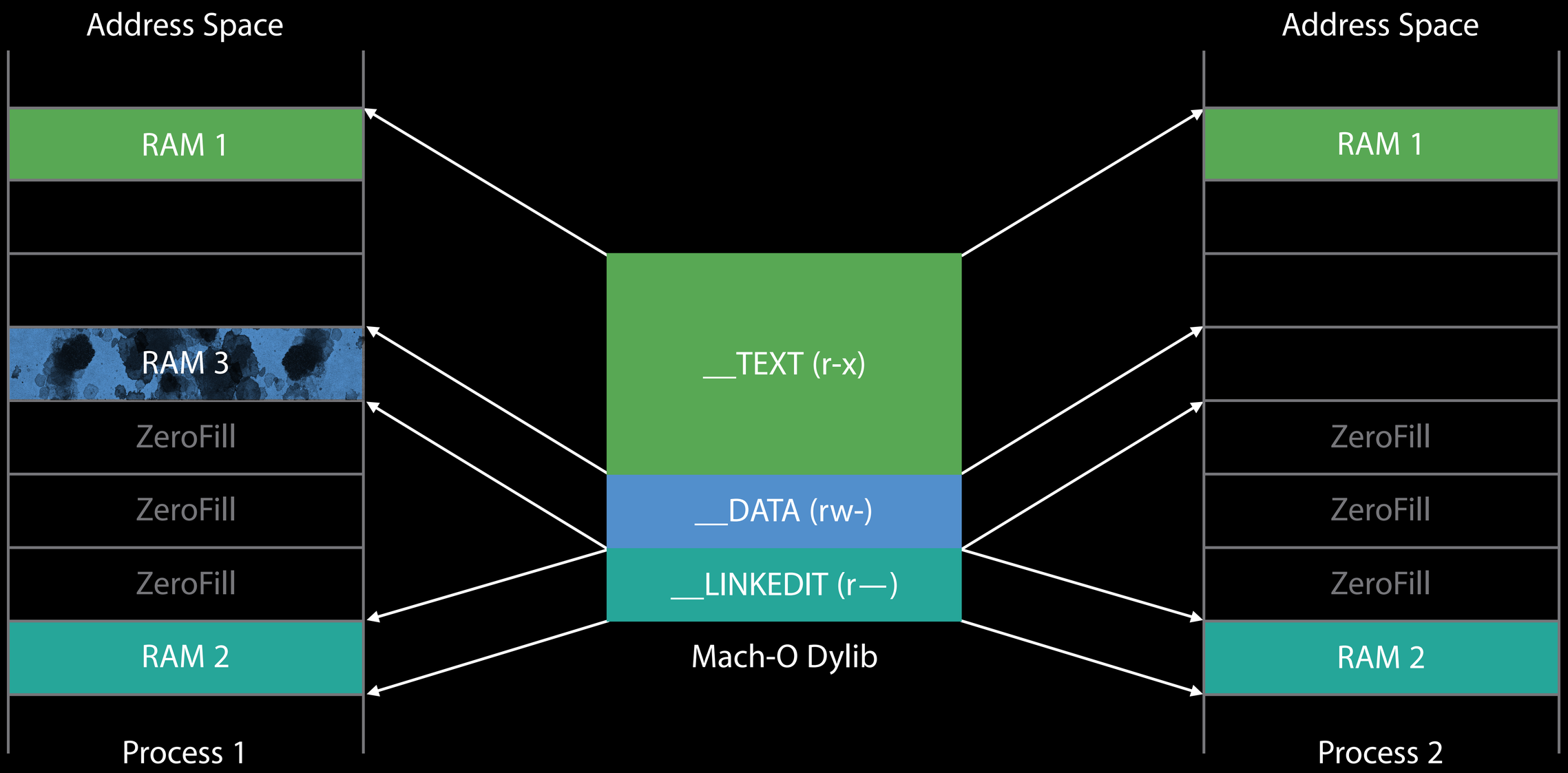
Mach-O Image Loading



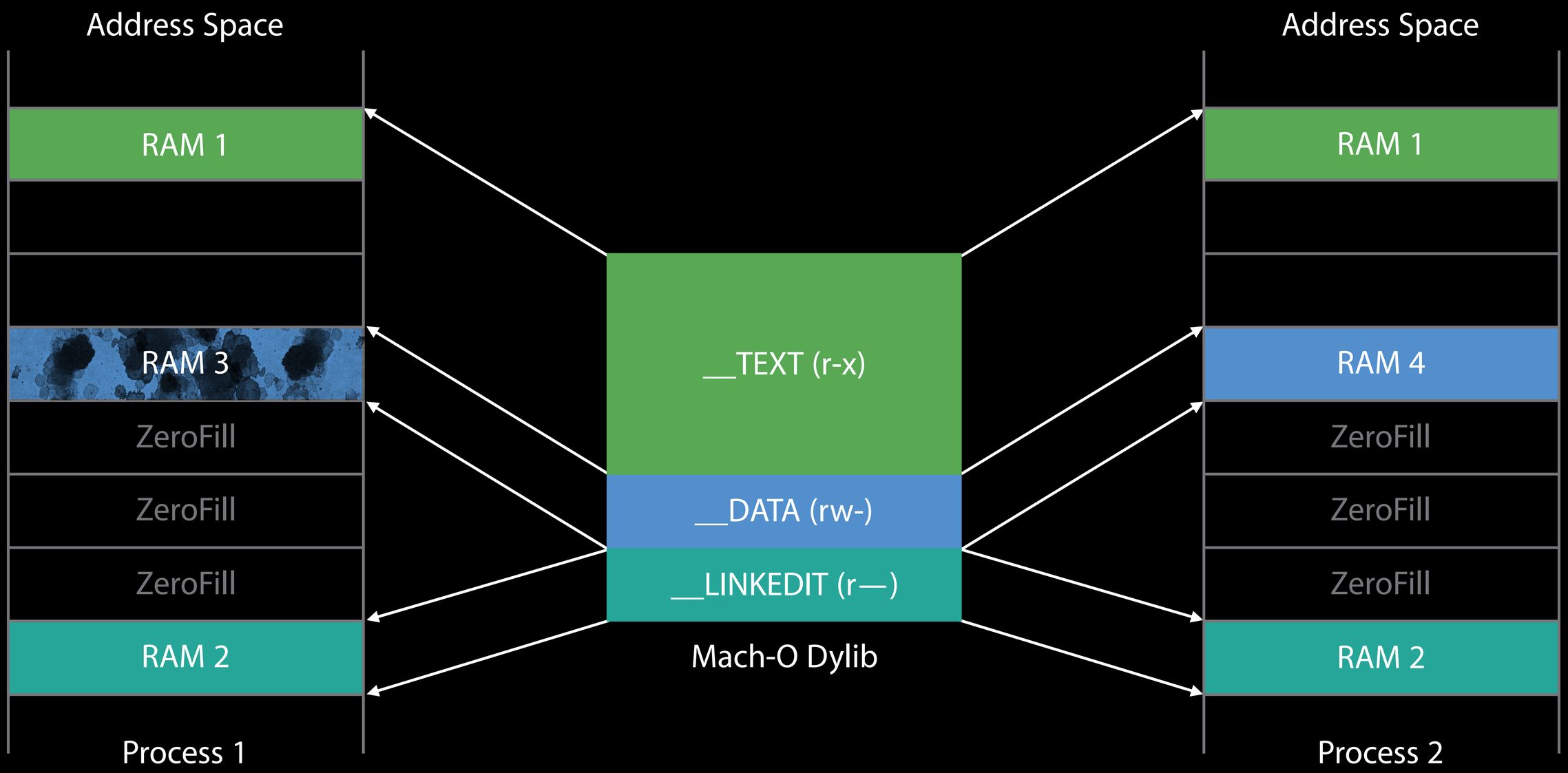
Mach-O Image Loading



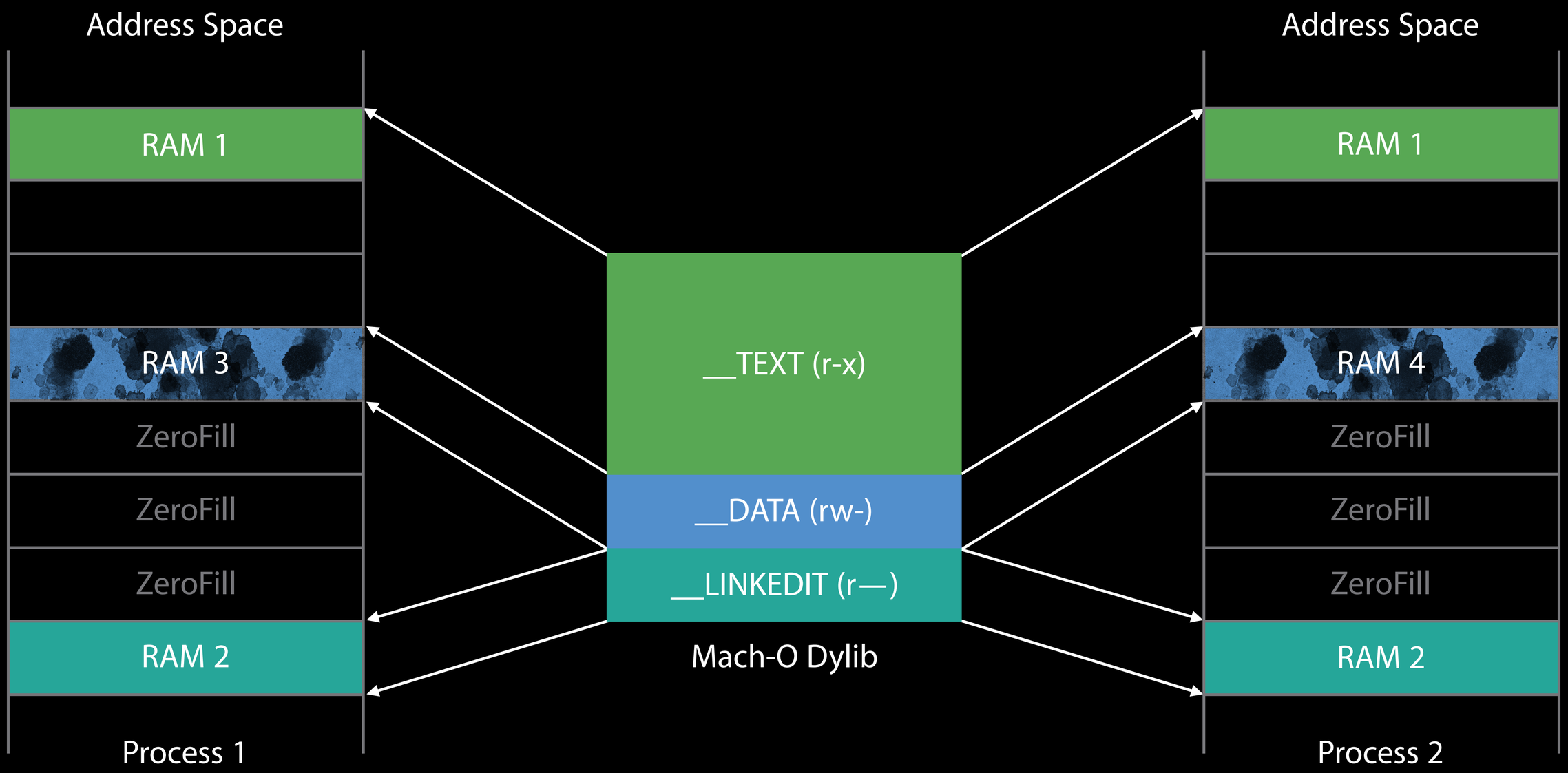
Mach-O Image Loading



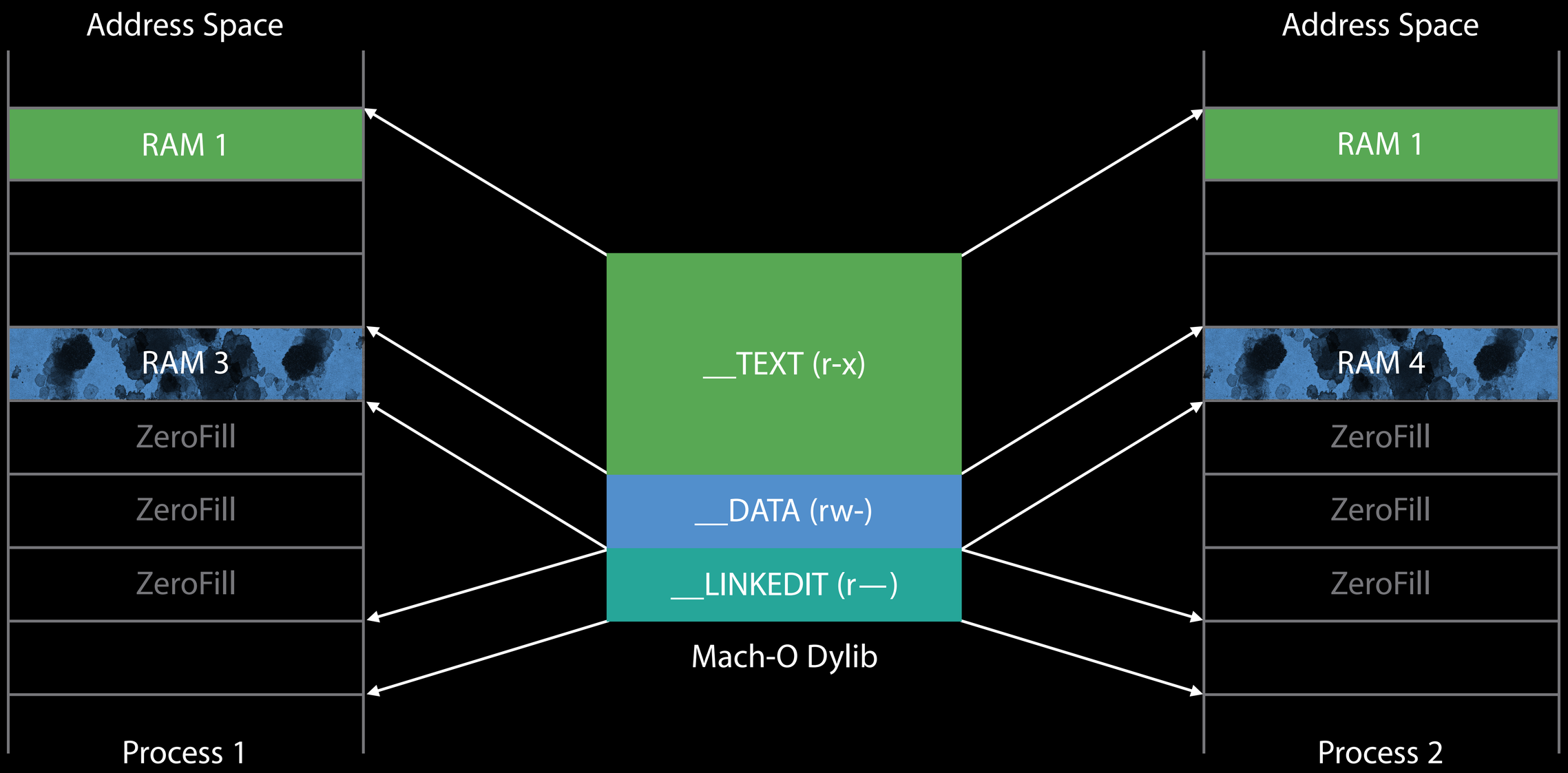
Mach-O Image Loading



Mach-O Image Loading



Mach-O Image Loading



Security

ASLR

- Address Space Layout Randomization
- Images load at random address

Security

ASLR

- Address Space Layout Randomization
- Images load at random address

Code Signing

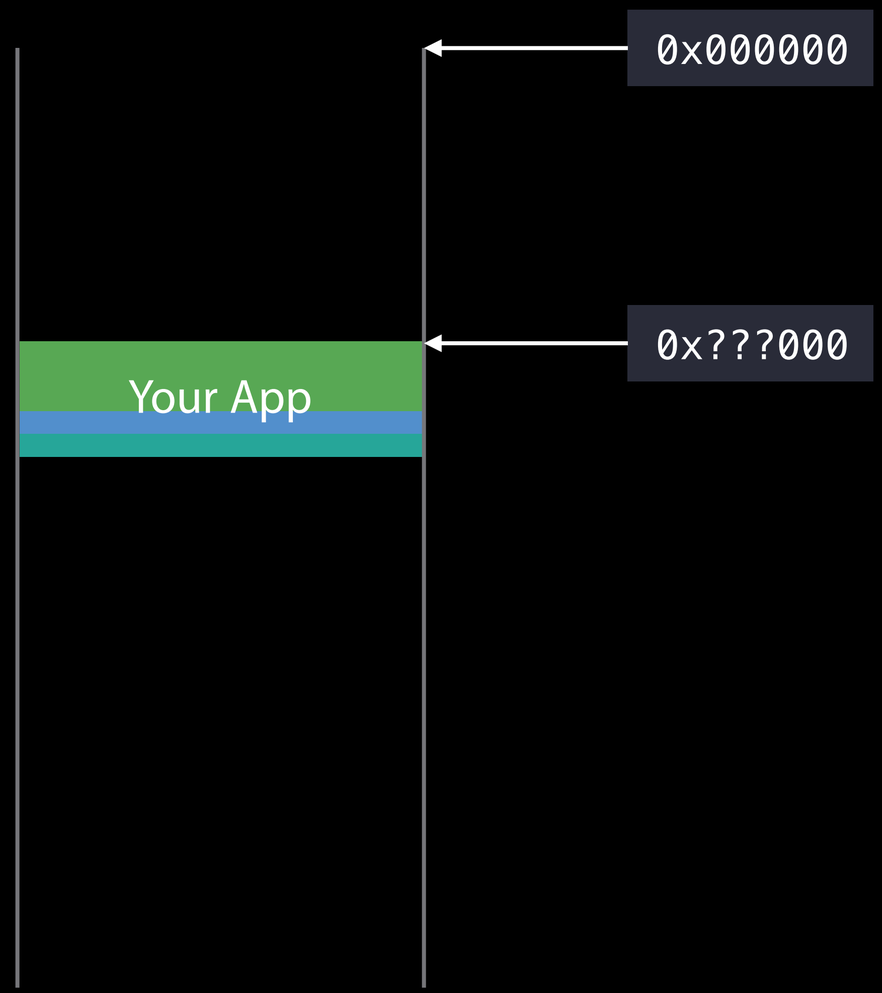
- Content of each page is hashed
- Hash is verified on page-in

exec() to main()

exec()

Kernel maps your application into new address space

Start of your app is random



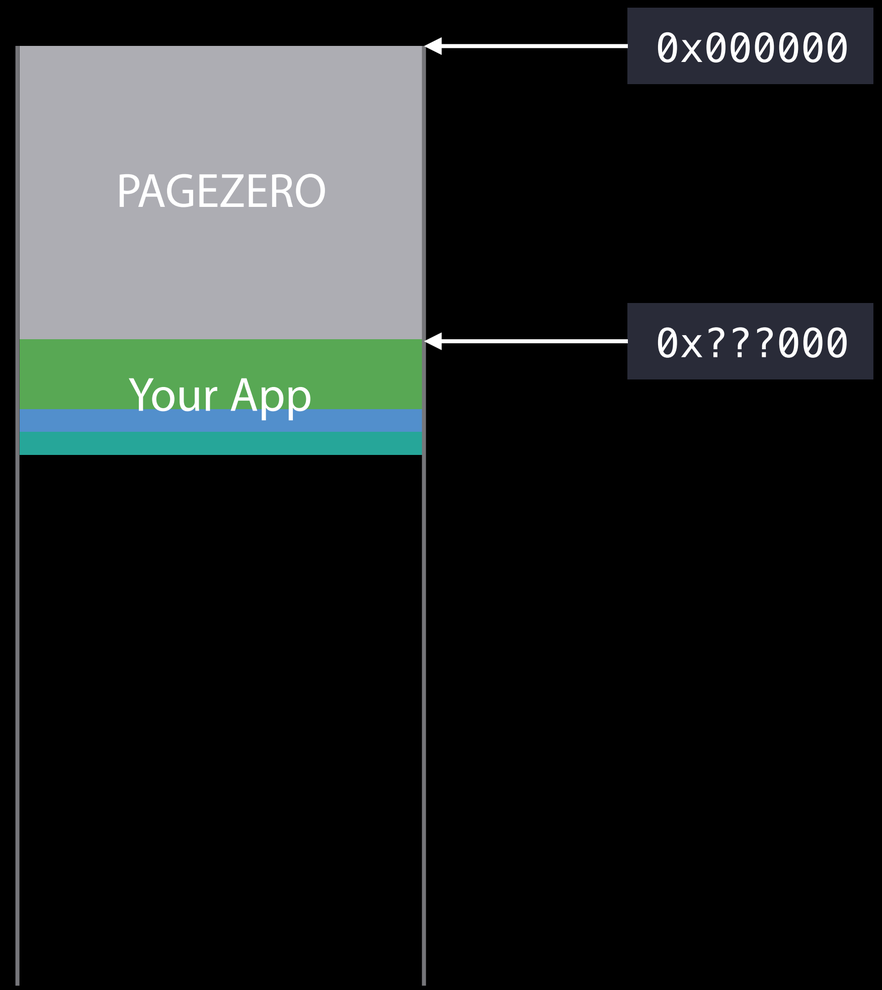
exec()

Kernel maps your application into new address space

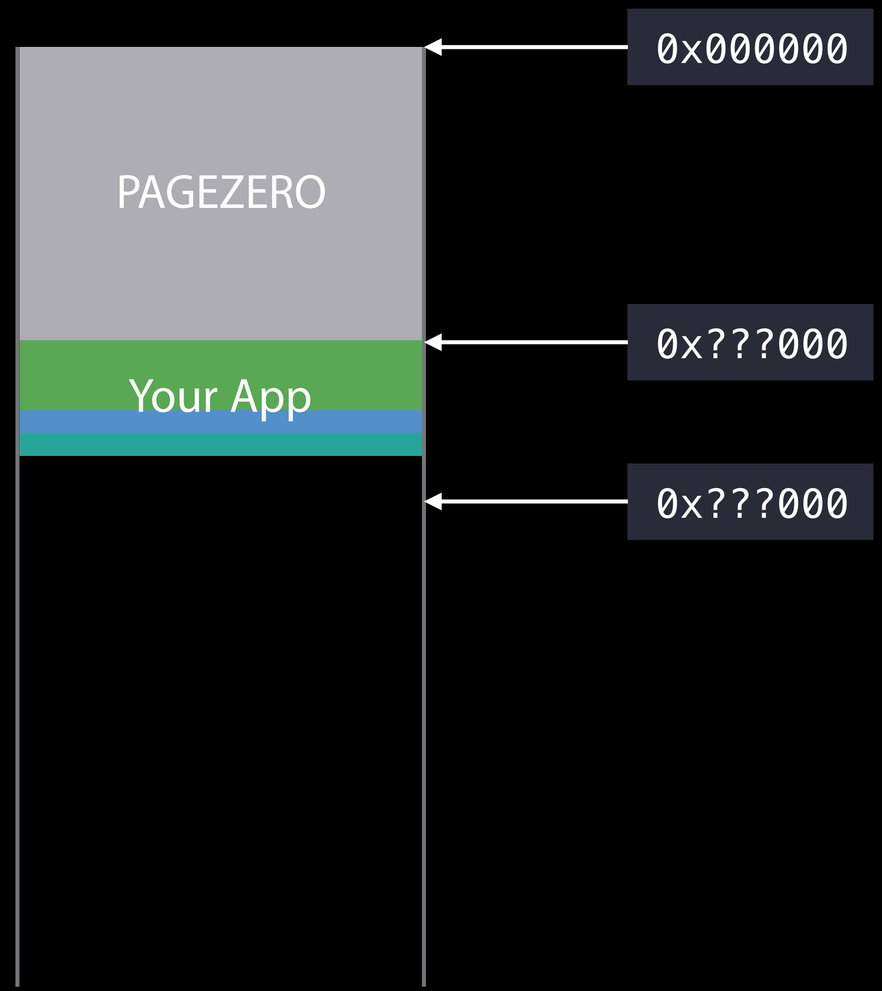
Start of your app is random

Low memory is marked inaccessible

- 4KB+ for 32-bit process
- 4GB+ for 64-bit processes
- Catches NULL pointer usage
- Catches pointer truncation errors



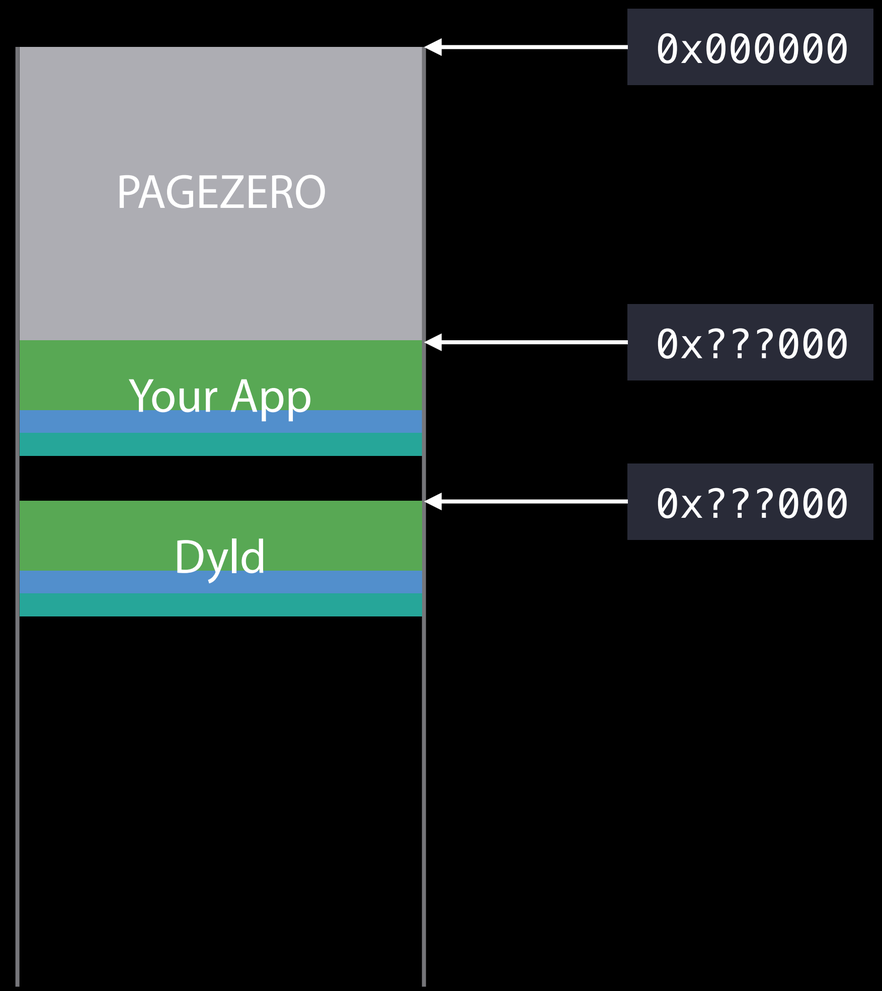
What About Dylibs?



What About Dylibs?

Kernel loads helper program

- Dyld (dynamic loader)
- Executions starts in dyld



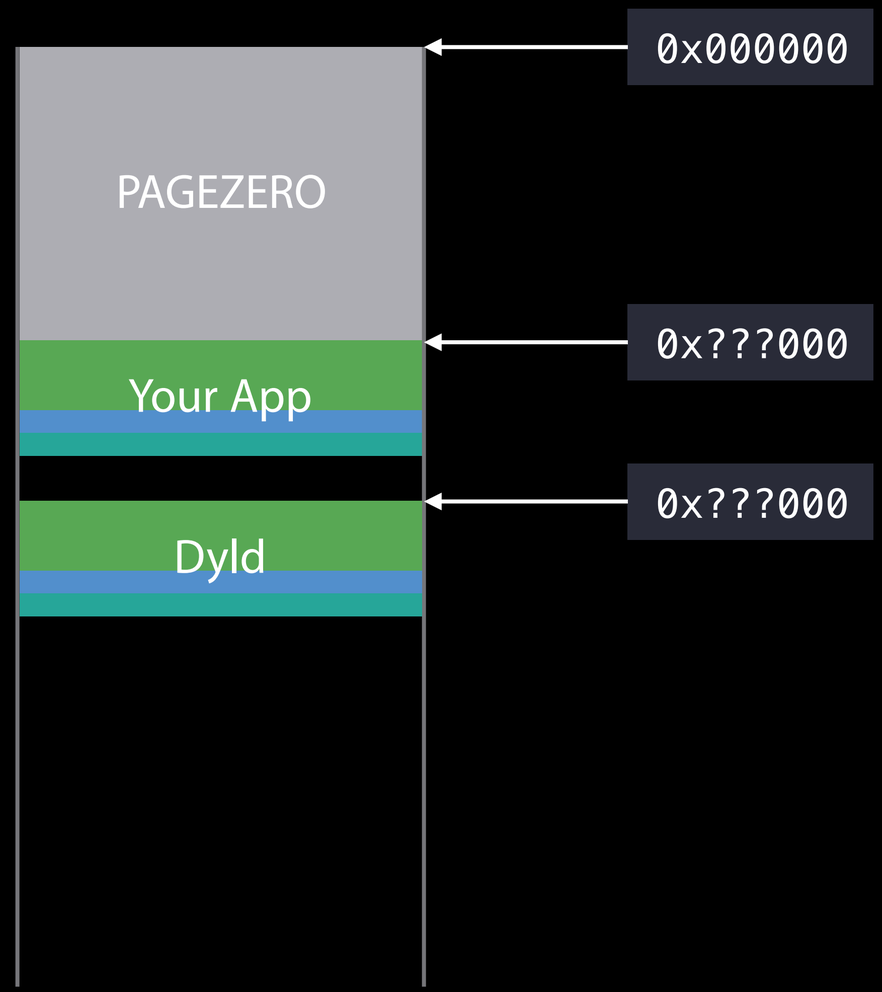
What About Dylibs?

Kernel loads helper program

- Dyld (dynamic loader)
- Executions starts in dyld

Dyld runs in-process

- Loads dependent dylibs
- Has same permissions as app



Dyld Steps

Dyld Steps

Map all dependent dylibs, recurse

Rebase all images

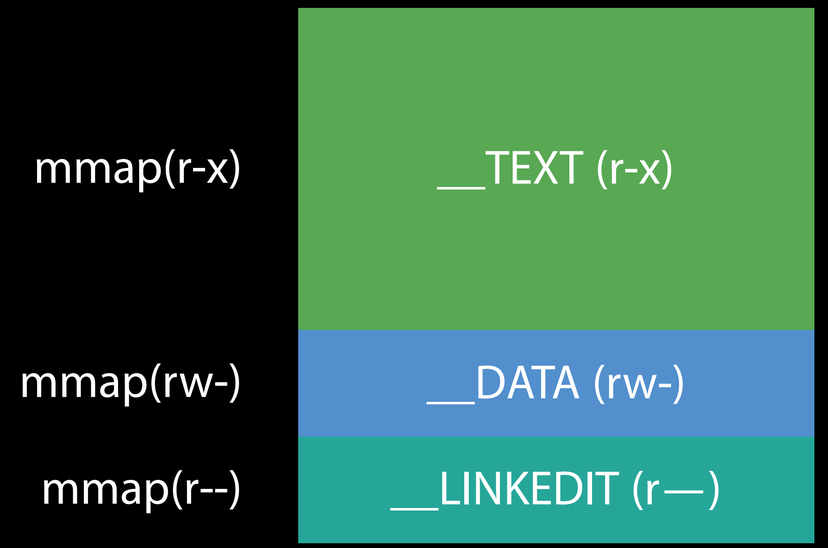
Bind all images

ObjC prepare images

Run initializers

Loading Dylibs

Parse list of dependent dylibs

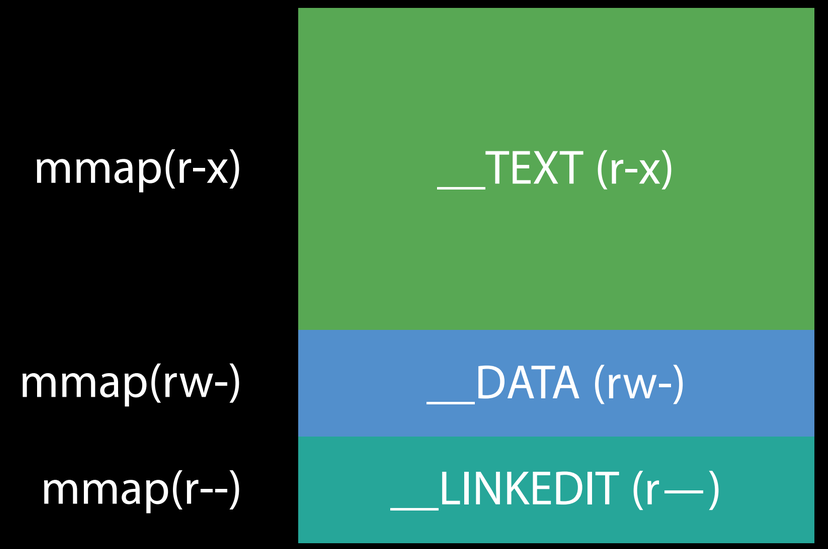


Load dylibs → Rebase → Bind → ObjC → Initializers

Loading Dylibs

Parse list of dependent dylibs

Find requested mach-o file

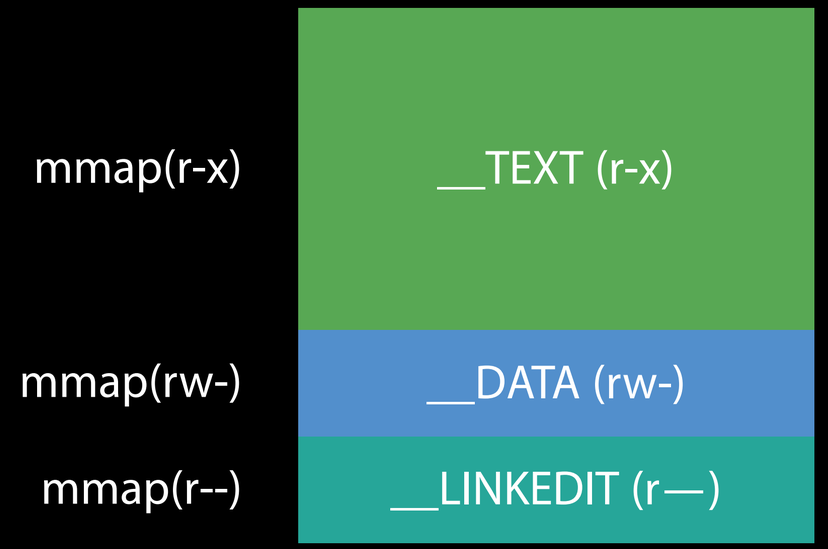


Loading Dylibs

Parse list of dependent dylibs

Find requested mach-o file

Open and read start of file



Load dylibs → Rebase → Bind → ObjC → Initializers

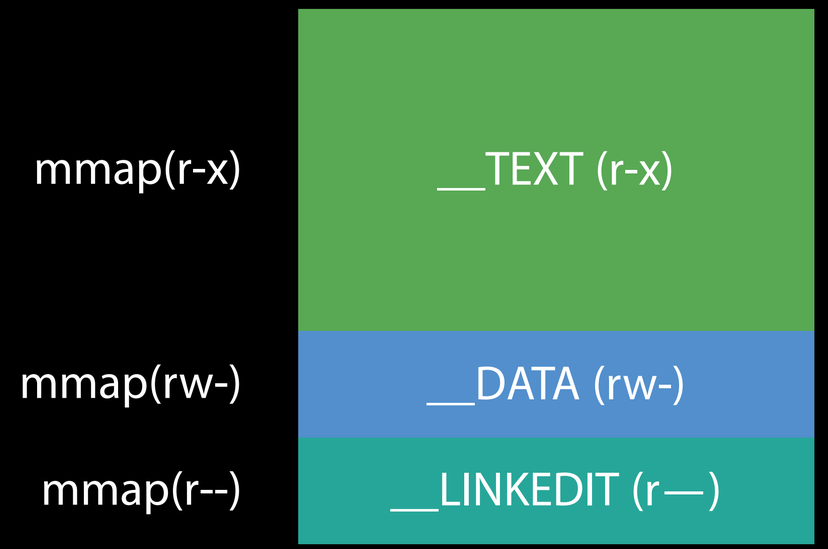
Loading Dylibs

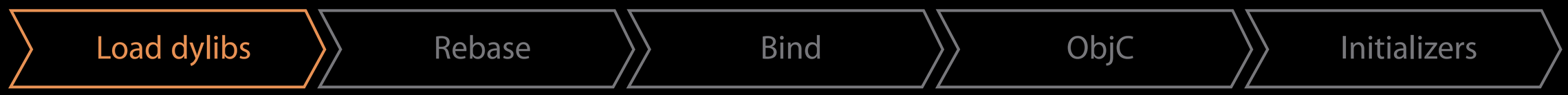
Parse list of dependent dylibs

Find requested mach-o file

Open and read start of file

Validate mach-o





Loading Dylibs

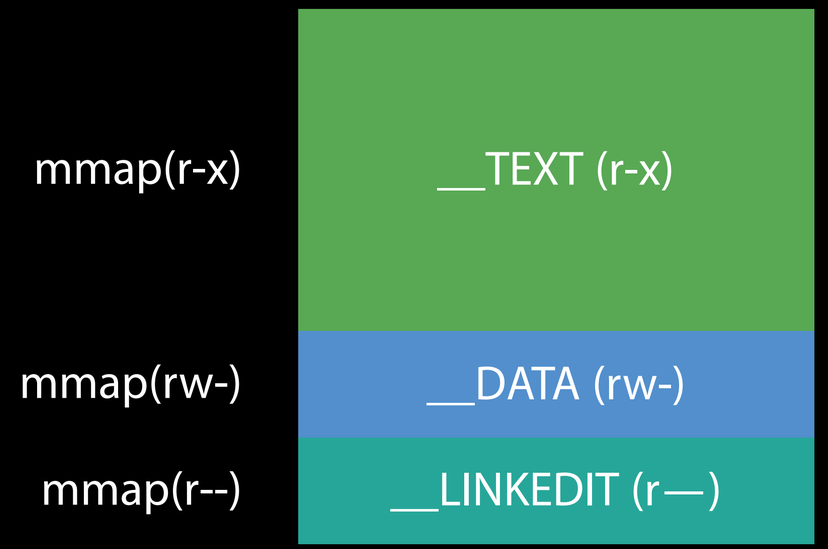
Parse list of dependent dylibs

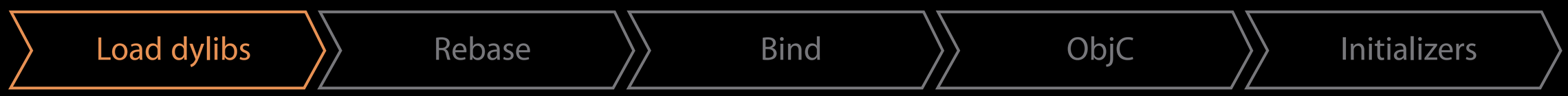
Find requested mach-o file

Open and read start of file

Validate mach-o

Register code signature





Loading Dylibs

Parse list of dependent dylibs

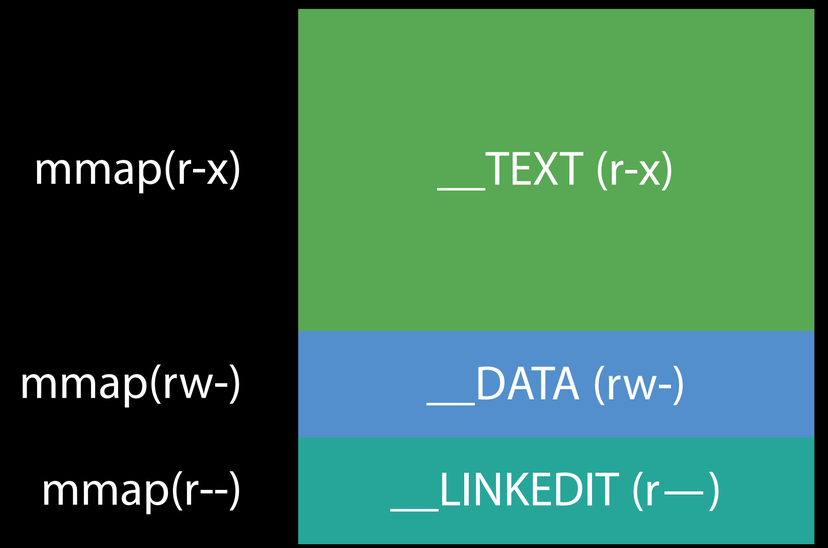
Find requested mach-o file

Open and read start of file

Validate mach-o

Register code signature

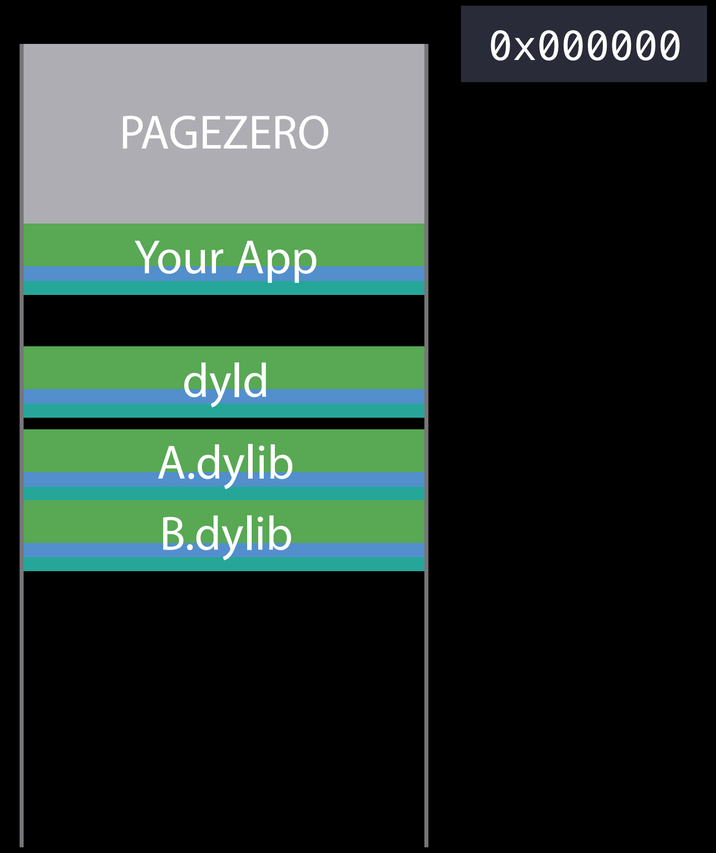
Call `mmap()` for each segment



Load dylibs → Rebase → Bind → ObjC → Initializers

Recursive Loading

All your app's direct dependents are loaded

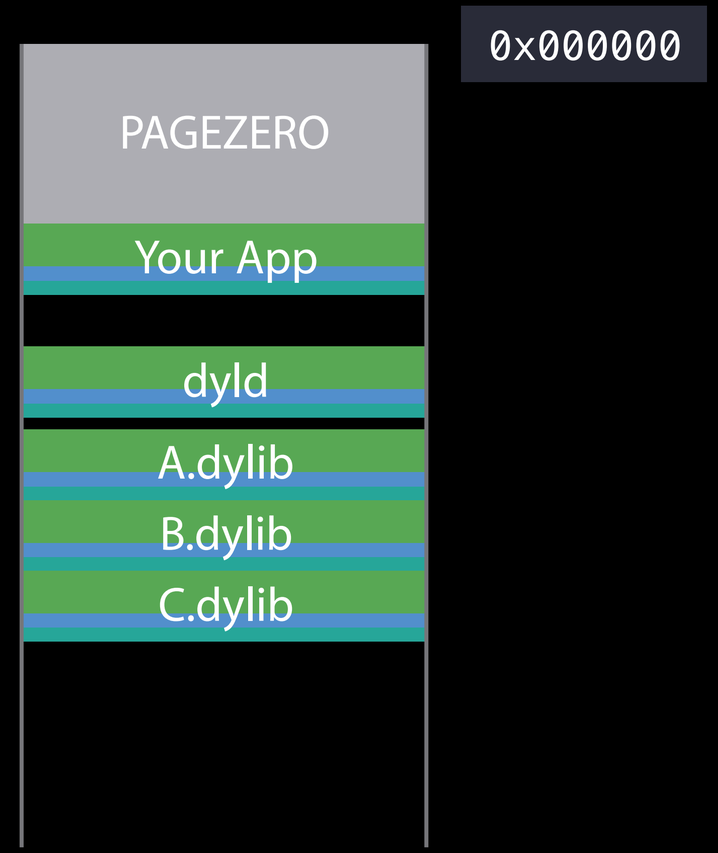


Load dylibs → Rebase → Bind → ObjC → Initializers

Recursive Loading

All your app's direct dependents are loaded

Plus any dylib's needed by those dylibs



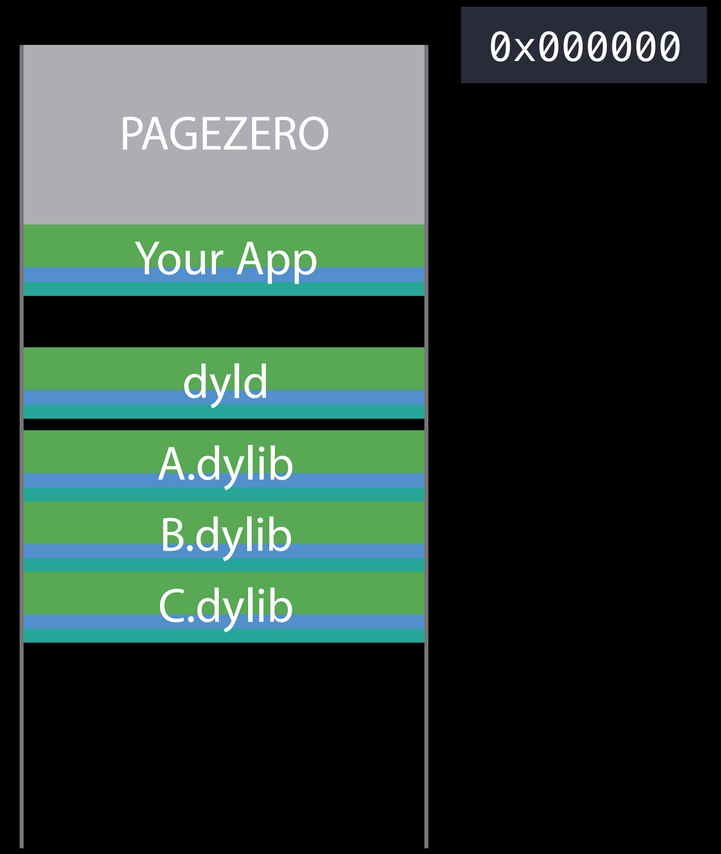
Load dylibs → Rebase → Bind → ObjC → Initializers

Recursive Loading

All your app's direct dependents are loaded

Plus any dylib's needed by those dylibs

Rinse and repeat



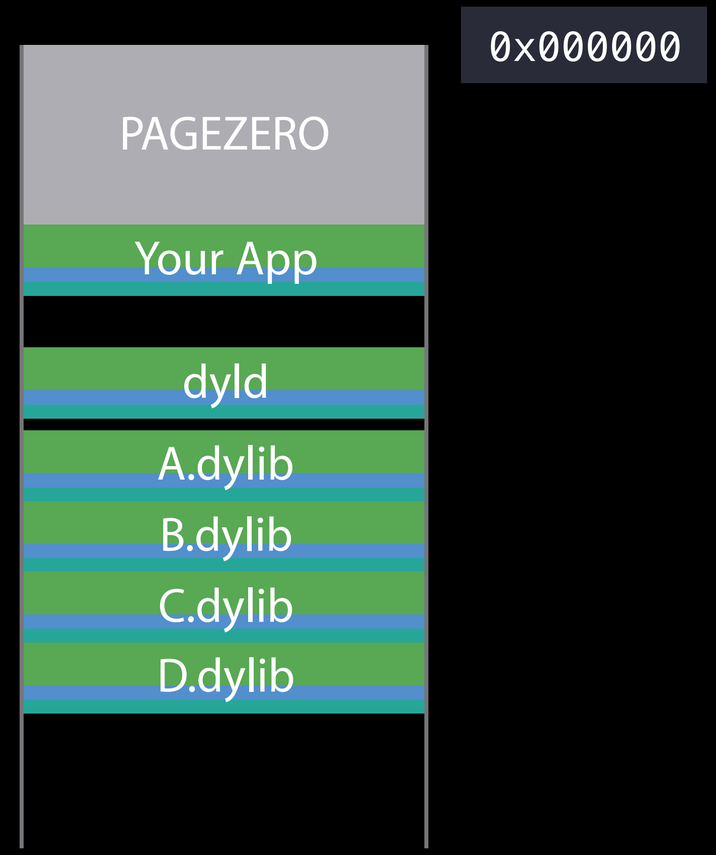
Load dylibs → Rebase → Bind → ObjC → Initializers

Recursive Loading

All your app's direct dependents are loaded

Plus any dylib's needed by those dylibs

Rinse and repeat



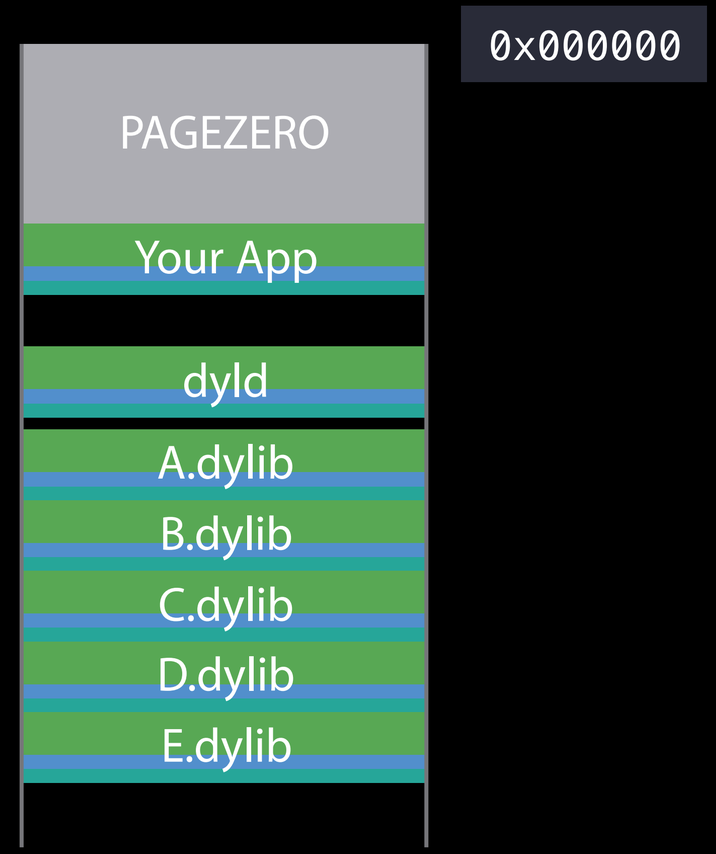
Load dylibs → Rebase → Bind → ObjC → Initializers

Recursive Loading

All your app's direct dependents are loaded

Plus any dylib's needed by those dylibs

Rinse and repeat



Load dylibs → Rebase → Bind → ObjC → Initializers

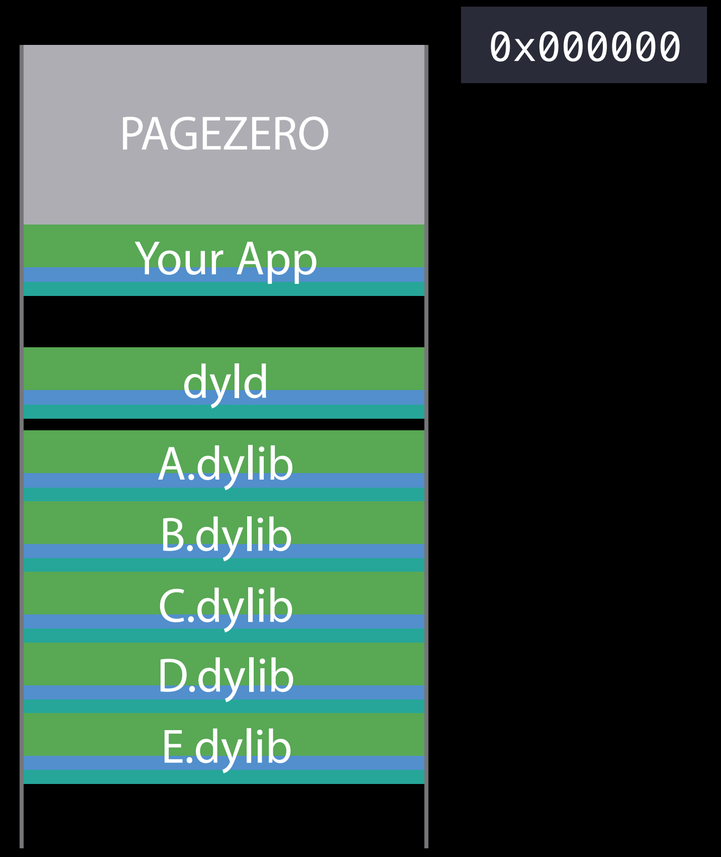
Recursive Loading

All your app's direct dependents are loaded

Plus any dylib's needed by those dylibs

Rinse and repeat

Apps typically load 100 to 400 dylibs!



Load dylibs | Rebase | Bind | ObjC | Initializers

Recursive Loading

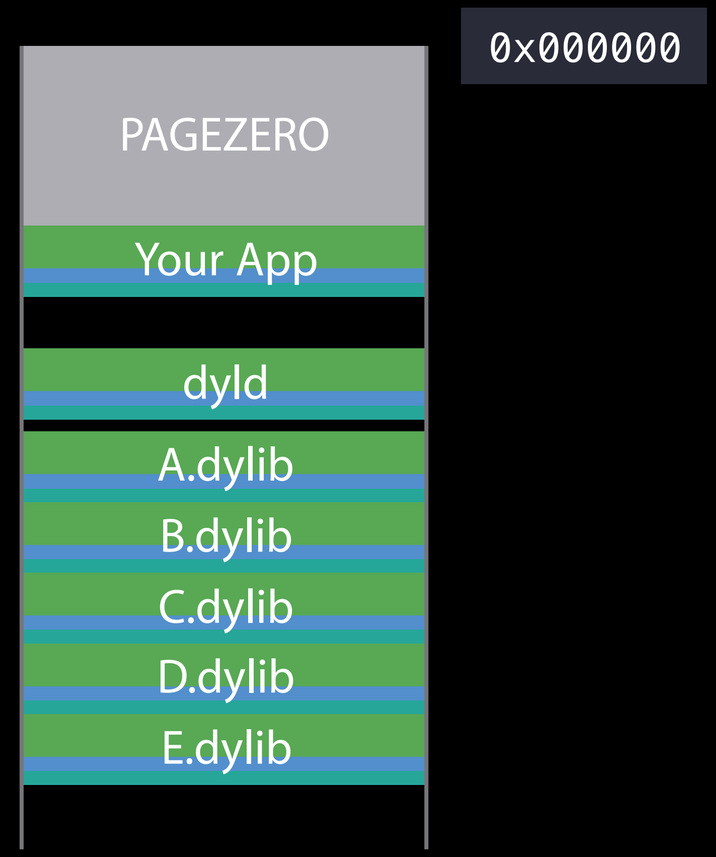
All your app's direct dependents are loaded

Plus any dylib's needed by those dylibs

Rinse and repeat

Apps typically load 100 to 400 dylibs!

- Most are OS dylibs



Load dylibs | Rebase | Bind | ObjC | Initializers

Recursive Loading

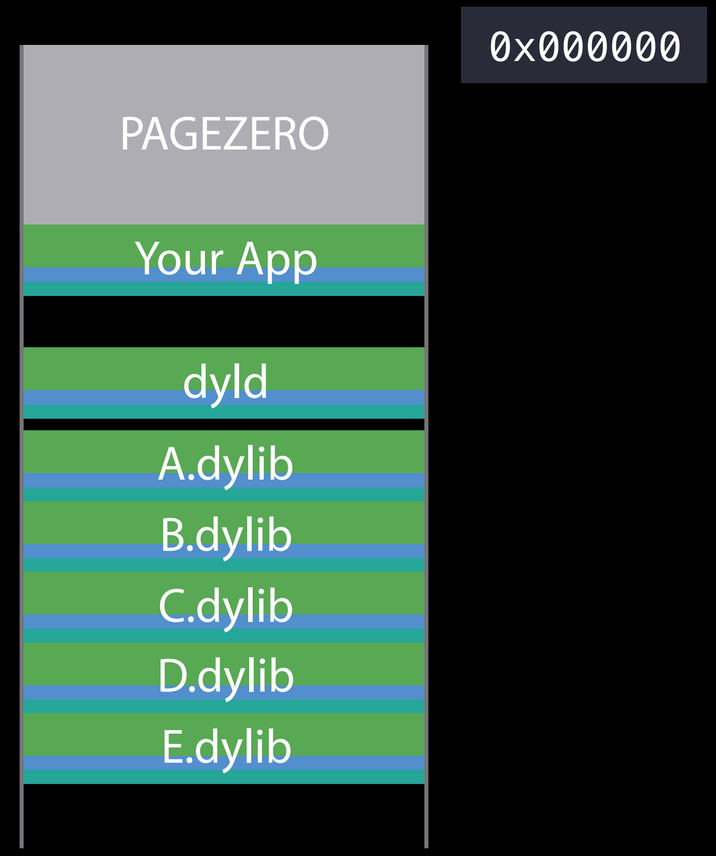
All your app's direct dependents are loaded

Plus any dylib's needed by those dylibs

Rinse and repeat

Apps typically load 100 to 400 dylibs!

- Most are OS dylibs
- We've optimized loading of OS dylibs



Load dylibs | Rebase | Bind | ObjC | Initializers

Fix-ups

Code signing means instructions cannot be altered

Fix-ups

Code signing means instructions cannot be altered

Modern code-gen is dynamic PIC (Position Independent Code)

Fix-ups

Code signing means instructions cannot be altered

Modern code-gen is dynamic PIC (Position Independent Code)

- Code can run loaded at any address and is never altered

Fix-ups

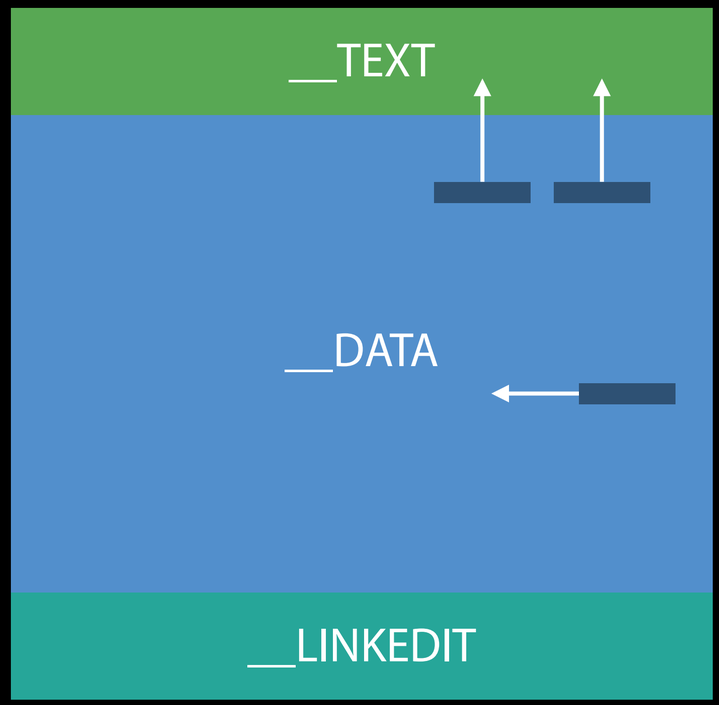
Code signing means instructions cannot be altered

Modern code-gen is dynamic PIC (Position Independent Code)

- Code can run loaded at any address and is never altered
- Instead, all fix ups are in __DATA

Rebasing and Binding

Rebasing: Adjusting pointers to within an image

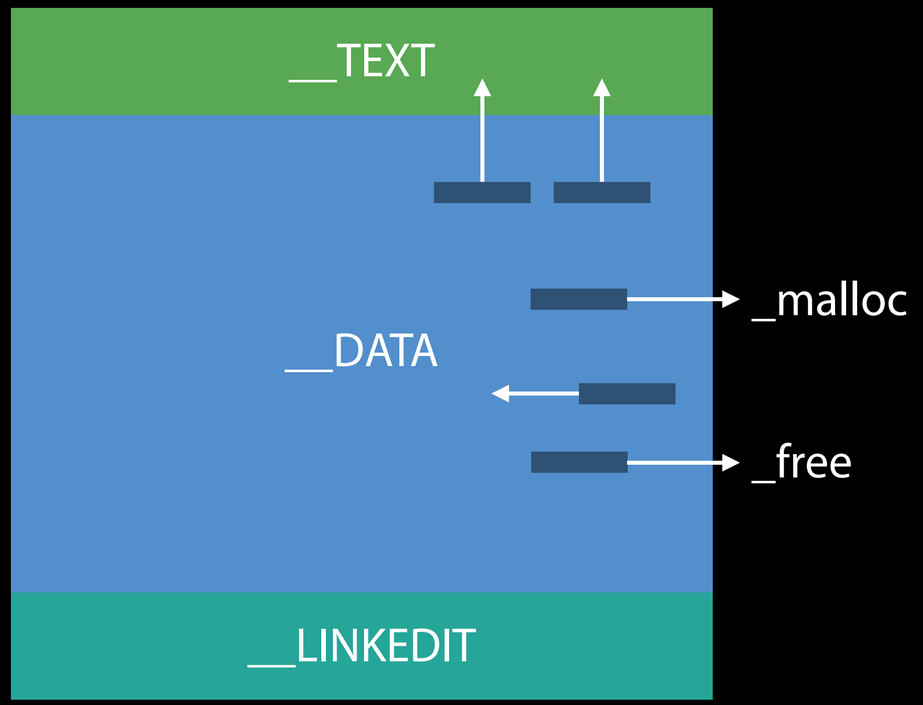


Load dylibs → Rebase → Bind → ObjC → Initializers

Rebasing and Binding

Rebasing: Adjusting pointers to within an image

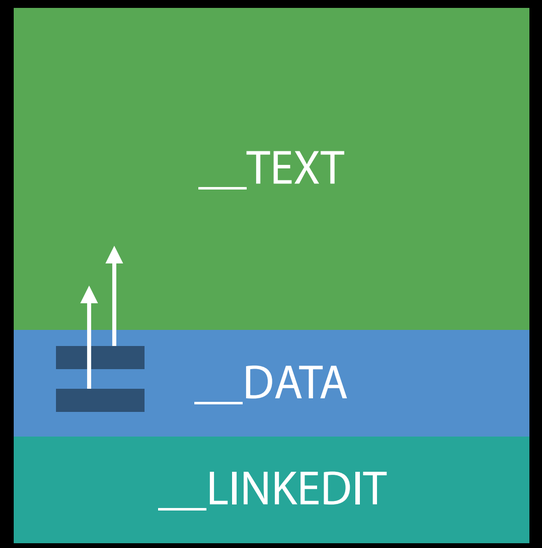
Binding: Setting pointers to outside image



```
[~]> xcrun dyldinfo -rebase -bind -lazy_bind myapp.app/myapp
rebase information:
segment section          address     type
__DATA  __const          0x10000C1A0  pointer
__DATA  __const          0x10000C1C0  pointer
__DATA  __const          0x10000C1E0  pointer
__DATA  __const          0x10000C210  pointer
…
bind information:
segment section          address      type   add dylib          symbol
__DATA  __objc_classrefs 0x10000D1E8  pointer  0 CoreFoundation  _OBJC_CLASS_$_NSObject
__DATA  __data           0x10000D4D0  pointer  0 CoreFoundation  _OBJC_METACLASS_$_NSObject
__DATA  __data           0x10000D558  pointer  0 CoreFoundation  _OBJC_METACLASS_$_NSObject
__DATA  __got            0x10000C018  pointer  0 libswiftCore    __TMSS
…
lazy binding information:
segment section          address     index  dylib           symbol
__DATA  __la_symbol_ptr  0x10000C0A8 0x0000 libSystem        __Block_copy
__DATA  __la_symbol_ptr  0x10000C0B0 0x0014 libSystem        __Block_release
__DATA  __la_symbol_ptr  0x10000C0B8 0x002B libSystem        _memcpy
…
```

Rebasing

Rebasing is adding a "slide" value to each internal pointer

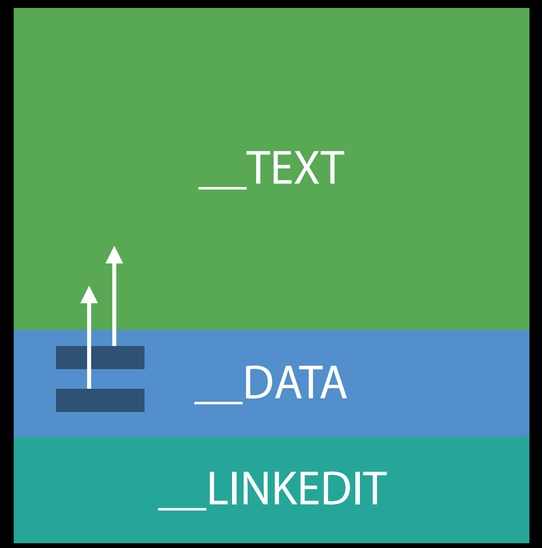


Load dylibs → **Rebase** → Bind → ObjC → Initializers

Rebasing

Rebasing is adding a "slide" value to each internal pointer

Slide = actual_address - preferred_address



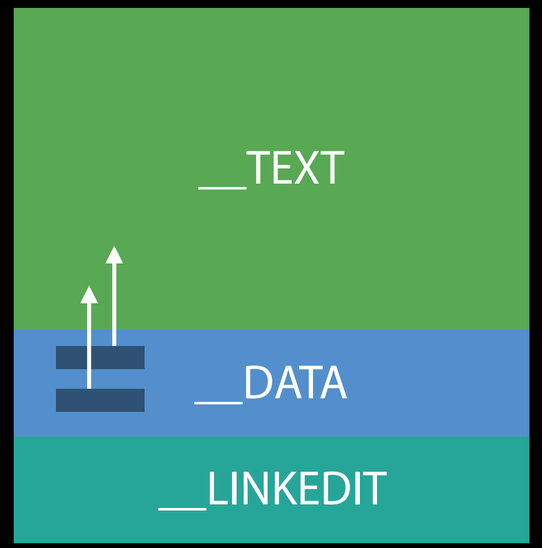
Load dylibs → Rebase → Bind → ObjC → Initializers

Rebasing

Rebasing is adding a "slide" value to each internal pointer

Slide = actual_address - preferred_address

Location of rebase locations is encoded in LINKEDIT



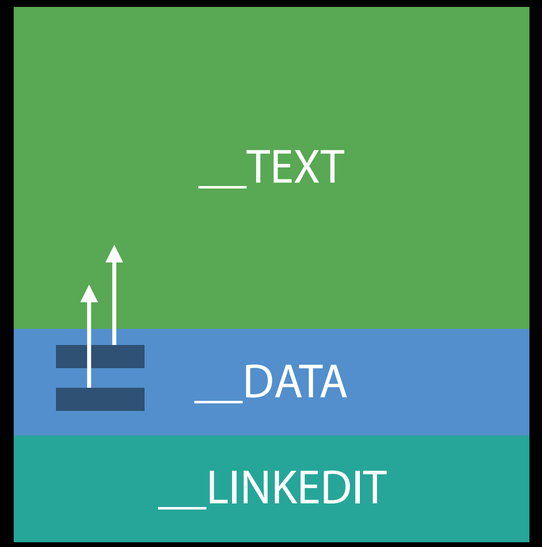
Rebasing

Rebasing is adding a "slide" value to each internal pointer

Slide = actual_address - preferred_address

Location of rebase locations is encoded in LINKEDIT

Pages-in and COW page



Rebasing

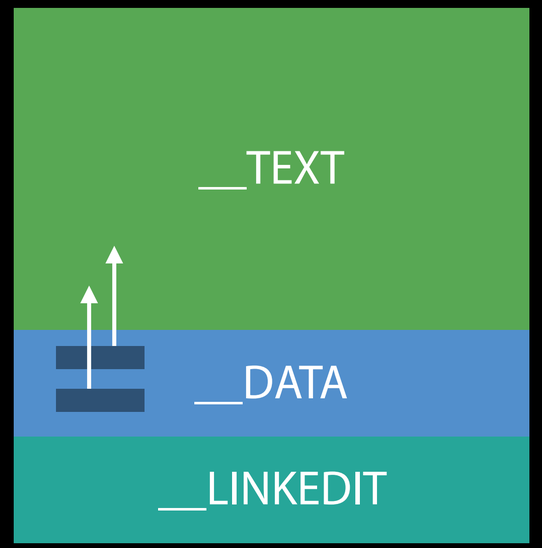
Rebasing is adding a "slide" value to each internal pointer

Slide = actual_address - preferred_address

Location of rebase locations is encoded in LINKEDIT

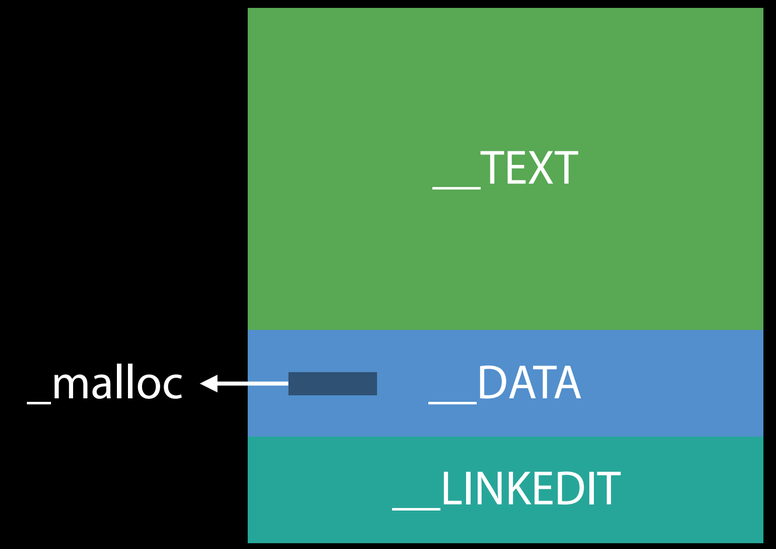
Pages-in and COW page

Rebasing is done in address order, so kernel starts prefetching



Binding

All references to something in another dylib are symbolic

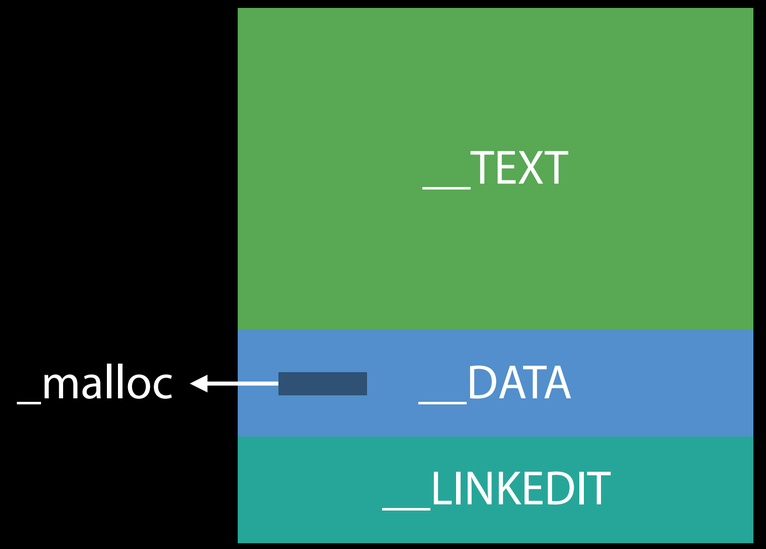


Load dylibs | Rebase | **Bind** | ObjC | Initializers

Binding

All references to something in another dylib are symbolic

Dyld needs to find symbol name



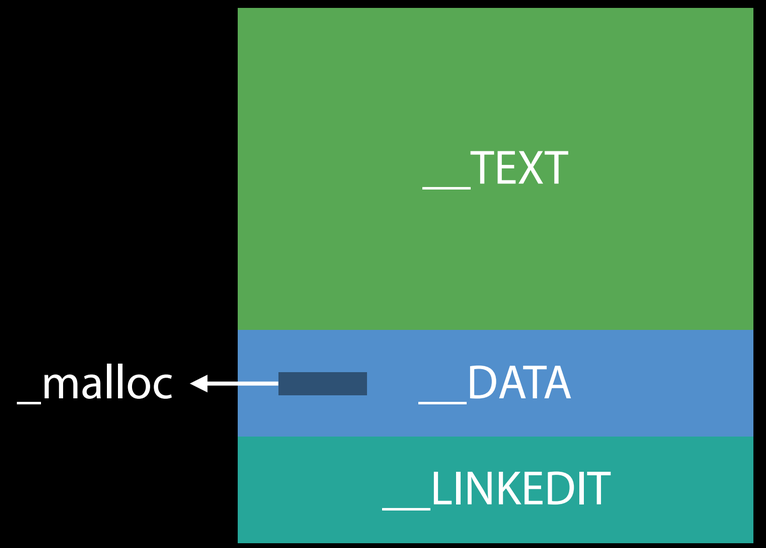
Load dylibs → Rebase → **Bind** → ObjC → Initializers

Binding

All references to something in another dylib are symbolic

Dyld needs to find symbol name

More computational than rebasing



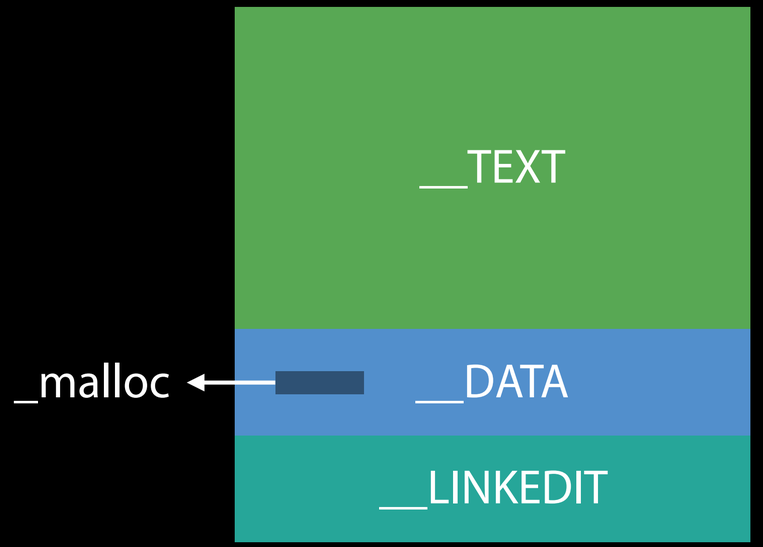
Binding

All references to something in another dylib are symbolic

Dyld needs to find symbol name

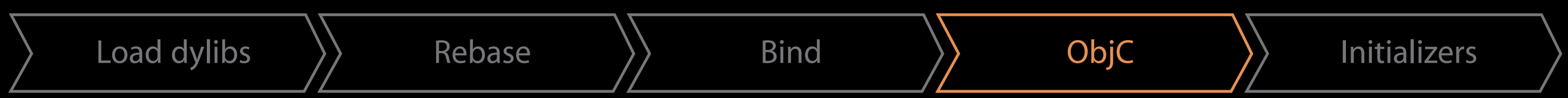
More computational than rebasing

Rarely page faults



Notify ObjC Runtime

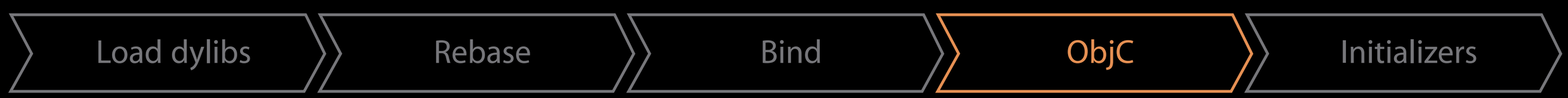
Most ObjC set up done via rebasing and binding



Notify ObjC Runtime

Most ObjC set up done via rebasing and binding

All ObjC class definitions are registered

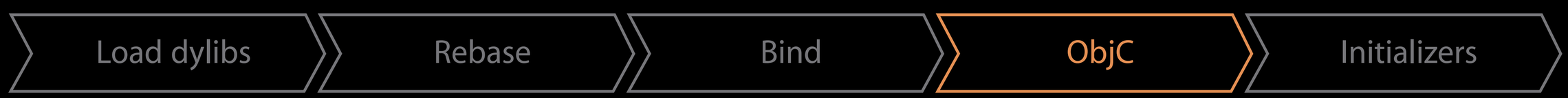


Notify ObjC Runtime

Most ObjC set up done via rebasing and binding

All ObjC class definitions are registered

Non-fragile ivars offsets updated



Notify ObjC Runtime

Most ObjC set up done via rebasing and binding

All ObjC class definitions are registered

Non-fragile ivars offsets updated

Categories are inserted into method lists

Notify ObjC Runtime

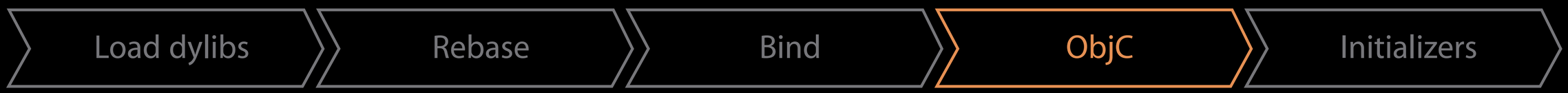
Most ObjC set up done via rebasing and binding

All ObjC class definitions are registered

Non-fragile ivars offsets updated

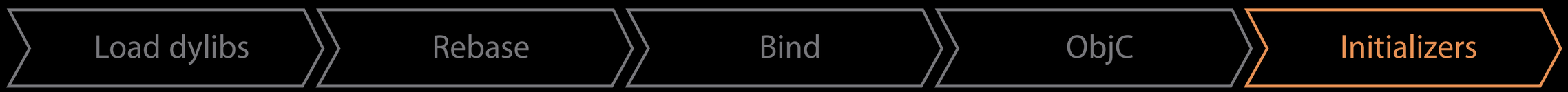
Categories are inserted into method lists

Selectors are uniqued



Initializers

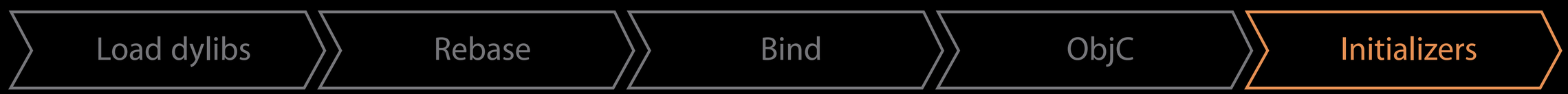
C++ generates initializer for statically allocated objects



Initializers

C++ generates initializer for statically allocated objects

ObjC +load methods



Initializers

C++ generates initializer for statically allocated objects

ObjC +load methods

Run "bottom up" so each initializer can call dylibs below it

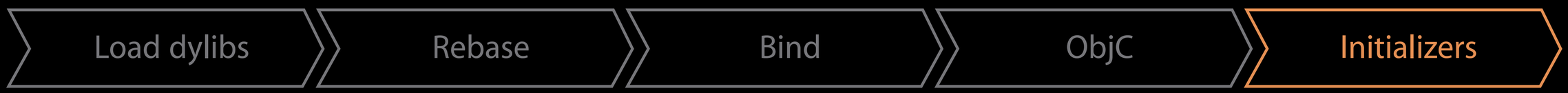
Initializers

C++ generates initializer for statically allocated objects

ObjC +load methods

Run "bottom up" so each initializer can call dylibs below it

Lastly, Dyld calls main() in executable



Pre-main() Summary

Dyld is a helper program

Pre-main() Summary

Dyld is a helper program

- Loads all dependent dylibs

Pre-main() Summary

Dyld is a helper program

- Loads all dependent dylibs
- Fixes up all pointers in DATA pages

Pre-main() Summary

Dyld is a helper program

- Loads all dependent dylibs
- Fixes up all pointers in DATA pages
- Runs all initializers

Putting Theory into Practice

Louis Gerbarg

Improving Launch Times

Overview

How fast?

Improving Launch Times

Overview

How fast?

How to measure?

Improving Launch Times

Overview

How fast?

How to measure?

Why is launch slow?

Improving Launch Times

Overview

How fast?

How to measure?

Why is launch slow?

What can you do?

Spoiler

Spoiler

Do Less Stuff

Improving Launch Times

Goals

Launch faster than animation

Improving Launch Times

Goals

Launch faster than animation

- Duration varies on devices

Improving Launch Times

Goals

Launch faster than animation

- Duration varies on devices
- 400ms is a good target

Improving Launch Times

Goals

Launch faster than animation

- Duration varies on devices
- 400ms is a good target

Don’t ever take longer than 20 seconds

Improving Launch Times

Goals

Launch faster than animation

- Duration varies on devices
- 400ms is a good target

Don’t ever take longer than 20 seconds

- App will be killed

Improving Launch Times

Goals

Launch faster than animation

- Duration varies on devices
- 400ms is a good target

Don’t ever take longer than 20 seconds

- App will be killed

Test on the slowest supported device

Improving Launch Times

Launch recap

Improving Launch Times

Launch recap

Parse images

Map images

Rebase images

Bind images

Run image initializers

Call `main()`

Improving Launch Times

Launch recap

Parse images

Map images

Rebase images

Bind images

Run image initializers

Call `main()`

Call `UIApplicationMain()`

Improving Launch Times

Launch recap

Parse images

Map images

Rebase images

Bind images

Run image initializers

Call `main()`

Call `UIApplicationMain()`

Call `applicationWillFinishLaunching`

Improving Launch Times

Warm vs. cold launch

Warm launch

Improving Launch Times

Warm vs. cold launch

Warm launch

- App and data already in memory

Improving Launch Times

Warm vs. cold launch

Warm launch

- App and data already in memory

Cold launch

Improving Launch Times

Warm vs. cold launch

Warm launch

- App and data already in memory

Cold launch

- App is not in kernel buffer cache

Improving Launch Times

Warm vs. cold launch

Warm launch

- App and data already in memory

Cold launch

- App is not in kernel buffer cache

Warm and cold launch times will be different

Improving Launch Times

Warm vs. cold launch

Warm launch

- App and data already in memory

Cold launch

- App is not in kernel buffer cache

Warm and cold launch times will be different

- Cold launch times are important

Improving Launch Times

Warm vs. cold launch

Warm launch

- App and data already in memory

Cold launch

- App is not in kernel buffer cache

Warm and cold launch times will be different

- Cold launch times are important
- Measure cold launch by rebooting

Improving Launch Times

Measurements

Measuring before `main()` is difficult

Improving Launch Times

Measurements

Measuring before `main()` is difficult

Dyld has built in measurements

Improving Launch Times

Measurements

Measuring before `main()` is difficult

Dyld has built in measurements

- `DYLD_PRINT_STATISTICS` environment variable

Improving Launch Times

Measurements

Measuring before `main()` is difficult

Dyld has built in measurements

- `DYLD_PRINT_STATISTICS` environment variable
  - Available on shipping OSes

Improving Launch Times

Measurements

NEW

Measuring before `main()` is difficult

Dyld has built in measurements

- `DYLD_PRINT_STATISTICS` environment variable
  - Available on shipping OSes
  - Significantly enhanced in new OSes

Improving Launch Times

Measurements

NEW

Measuring before `main()` is difficult

Dyld has built in measurements

- `DYLD_PRINT_STATISTICS` environment variable
  - Available on shipping OSes
  - Significantly enhanced in new OSes
  - Available in seed 2

Improving Launch Times

Measurements

NEW

Measuring before `main()` is difficult

Dyld has built in measurements

- `DYLD_PRINT_STATISTICS` environment variable
  - Available on shipping OSes
  - Significantly enhanced in new OSes
  - Available in seed 2

Debugger pauses every dylib load

Improving Launch Times

Measurements

NEW

Measuring before `main()` is difficult

Dyld has built in measurements

- `DYLD_PRINT_STATISTICS` environment variable
  - Available on shipping OSes
  - Significantly enhanced in new OSes
  - Available in seed 2

Debugger pauses every dylib load

- Dyld subtracts out debugger time

Improving Launch Times

Measurements

NEW

Measuring before `main()` is difficult

Dyld has built in measurements

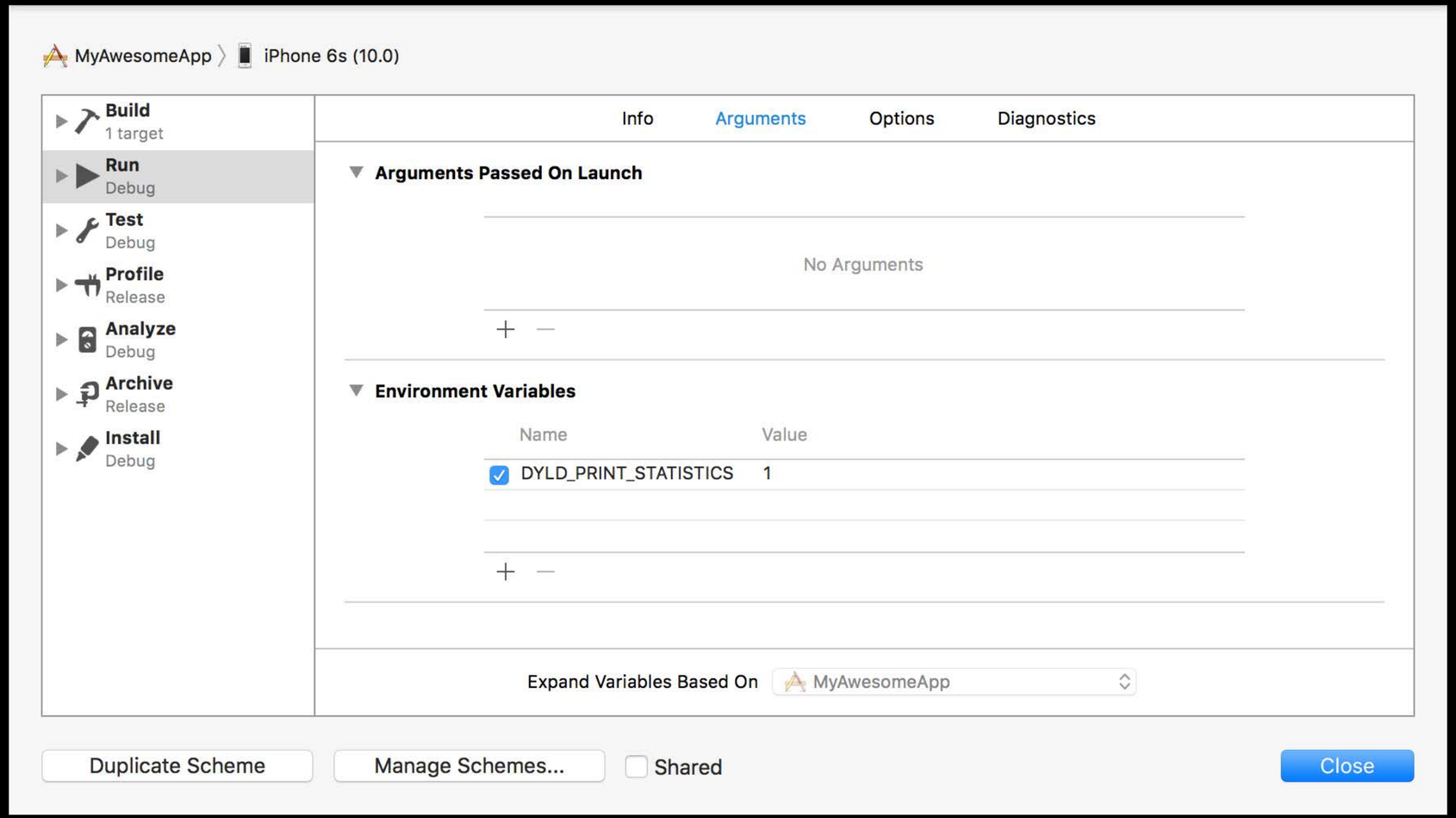
- `DYLD_PRINT_STATISTICS` environment variable
  - Available on shipping OSes
  - Significantly enhanced in new OSes
  - Available in seed 2

Debugger pauses every dylib load

- Dyld subtracts out debugger time
- Console times less than wall clock

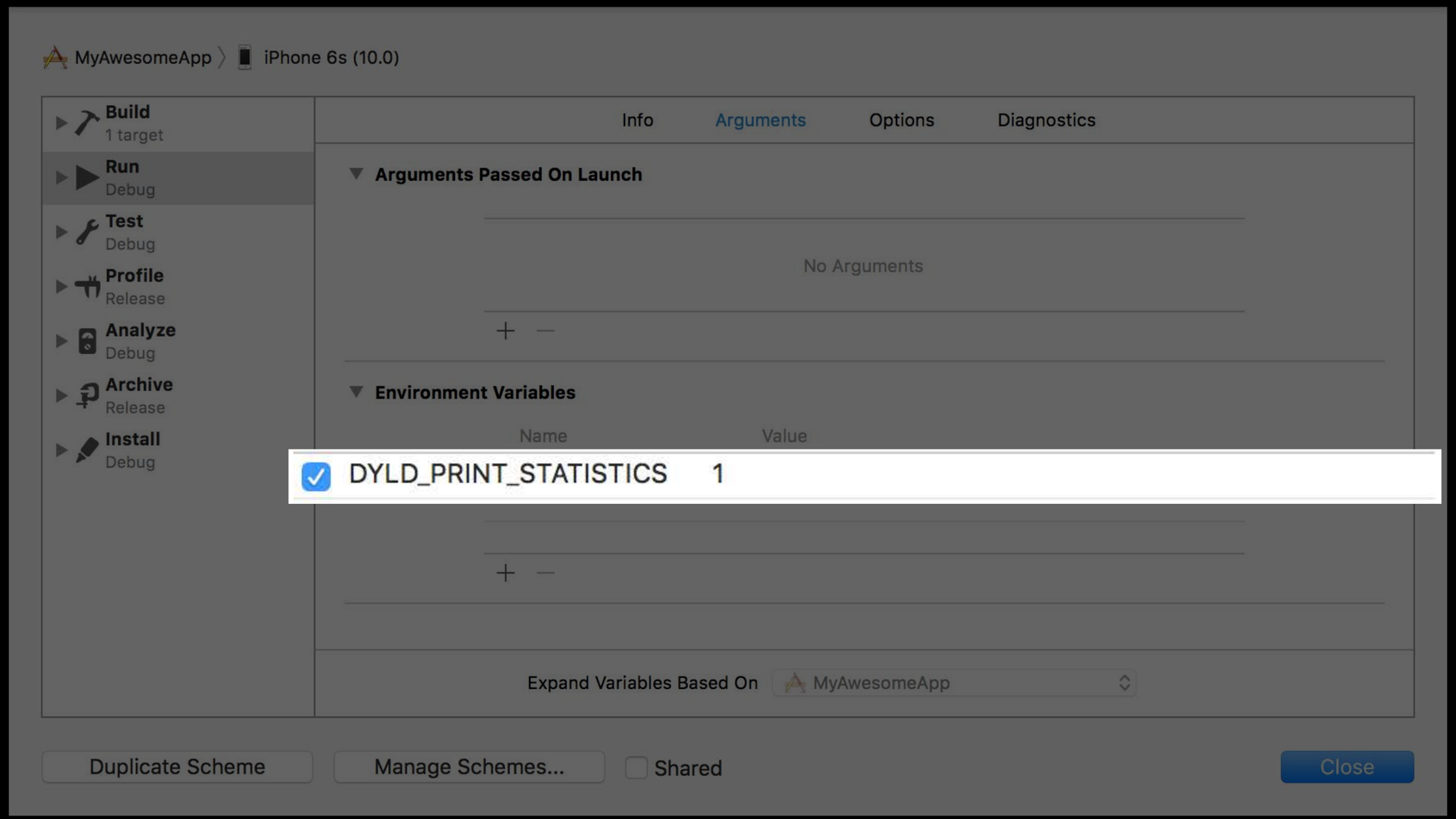
Improving Launch Times

DYLD_PRINT_STATISTICS



Improving Launch Times

DYLD_PRINT_STATISTICS



```
Total pre-main time: 10.6 seconds (100.0%)
         dylib loading time:  240.09 milliseconds (2.2%)
        rebase/binding time:  351.29 milliseconds (3.3%)
            ObjC setup time:  11.83 milliseconds (0.1%)
           initializer time:  10 seconds (94.3%)
          slowest intializers :
                MyAwesomeApp :  10.0 seconds (94.2%)
```

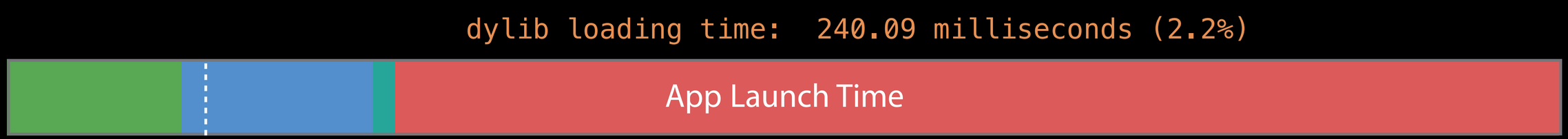
```
Total pre-main time: 10.6 seconds (100.0%)
         dylib loading time:  240.09 milliseconds (2.2%)
        rebase/binding time:  351.29 milliseconds (3.3%)
            ObjC setup time:  11.83 milliseconds (0.1%)
           initializer time:  10 seconds (94.3%)
          slowest intializers :
                MyAwesomeApp :  10.0 seconds (94.2%)
```

```
Total pre-main time: 10.6 seconds (100.0%)
         dylib loading time:  240.09 milliseconds (2.2%)
        rebase/binding time:  351.29 milliseconds (3.3%)
           ObjC setup time:  11.83 milliseconds (0.1%)
          initializer time:  10 seconds (94.3%)

      slowest intializers :
            MyAwesomeApp :  10.0 seconds (94.2%)
```

Dylib Loading

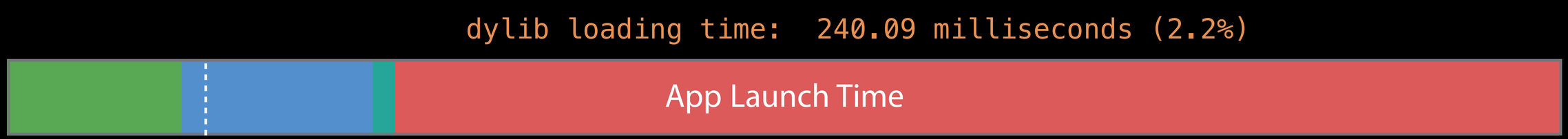
Embedded dylibs are expensive



Dylib Loading

Embedded dylibs are expensive

Use fewer dylibs

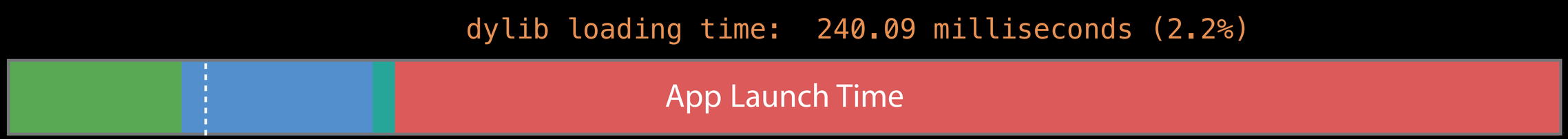


Dylib Loading

Embedded dylibs are expensive

Use fewer dylibs

- Merge existing dylibs

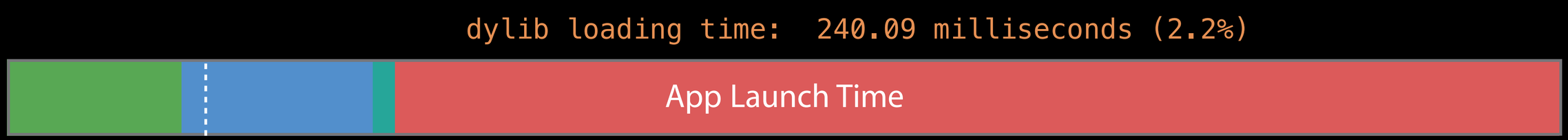


Dylib Loading

Embedded dylibs are expensive

Use fewer dylibs

- Merge existing dylibs
- Use static archives



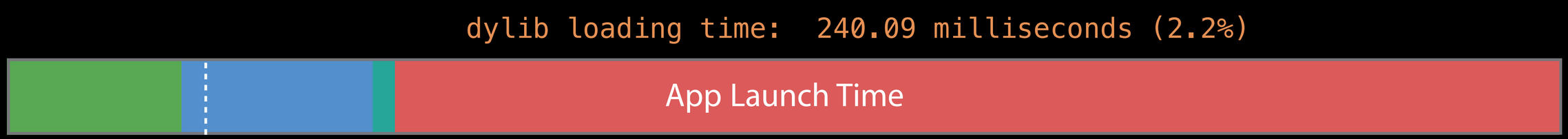
Dylib Loading

Embedded dylibs are expensive

Use fewer dylibs

- Merge existing dylibs
- Use static archives

Lazy load, but…



Dylib Loading

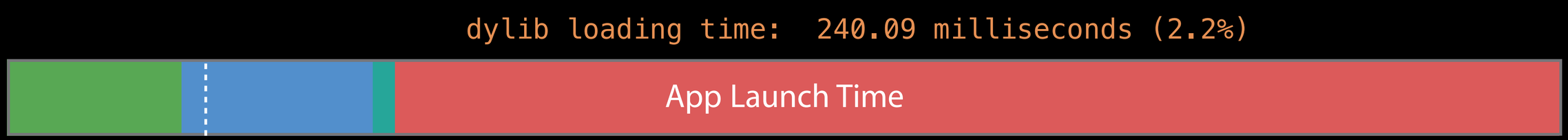
Embedded dylibs are expensive

Use fewer dylibs

- Merge existing dylibs
- Use static archives

Lazy load, but…

- `dlopen()` can cause issues
- Actually more work overall



Dylib Loading

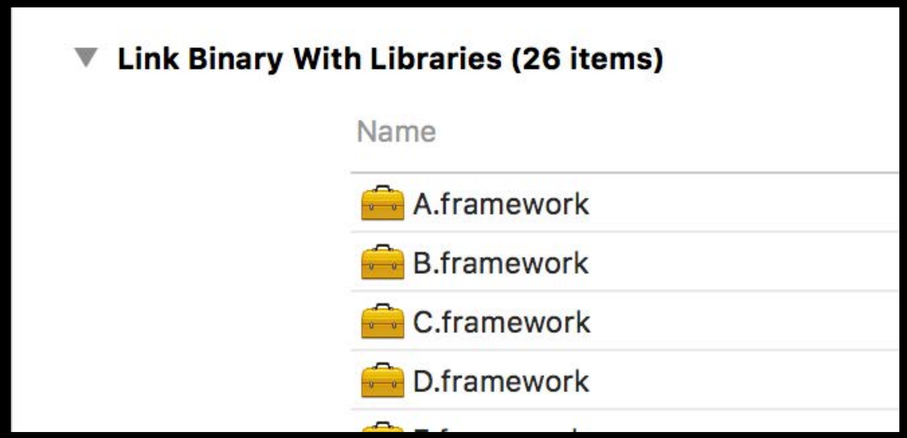
Embedded dylibs are expensive

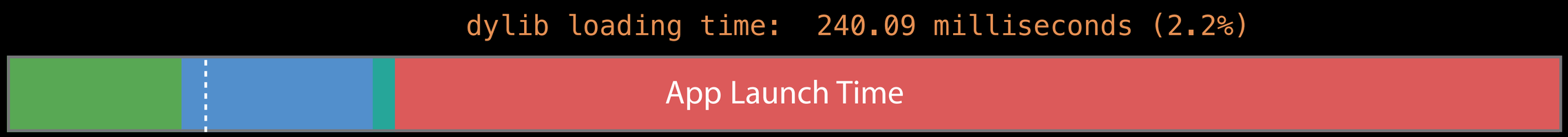
Use fewer dylibs

- Merge existing dylibs
- Use static archives

Lazy load, but…

- `dlopen()` can cause issues
- Actually more work overall





Dylib Loading

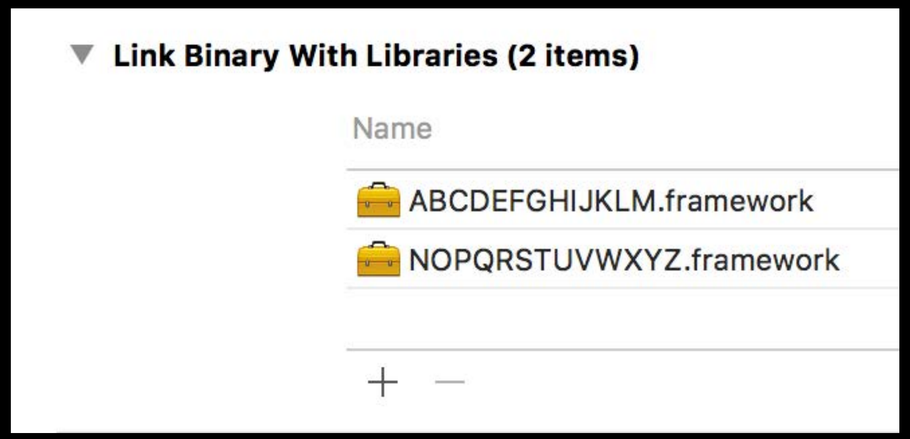
Embedded dylibs are expensive

Use fewer dylibs

- Merge existing dylibs
- Use static archives

Lazy load, but…

- `dlopen()` can cause issues
- Actually more work overall



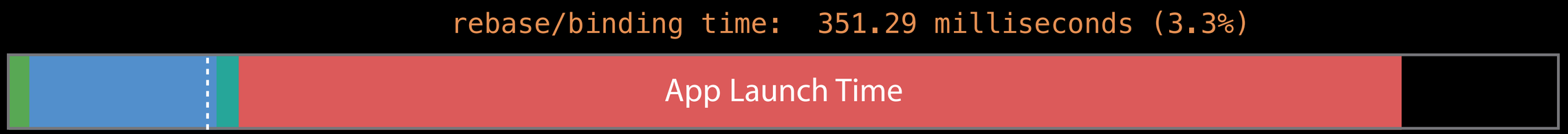

```
dylib loading time:  21.75 milliseconds (0.2%)
```

```
Total pre-main time: 10.4 seconds (100.0%)
         dylib loading time:  21.75 milliseconds (0.2%)
        rebase/binding time:  351.29 milliseconds (3.3%)
           ObjC setup time:  11.83 milliseconds (0.1%)
          initializer time:  10 seconds (94.3%)
     slowest intializers :
               MyAwesomeApp :  10.0 seconds (96.1%)
```

```
Total pre-main time: 10.4 seconds (100.0%)
         dylib loading time:  21.75 milliseconds (0.2%)
        rebase/binding time:  351.29 milliseconds (3.3%)
            ObjC setup time:  11.83 milliseconds (0.1%)
           initializer time:  10 seconds (94.3%)
           slowest intializers :
             MyAwesomeApp :  10.0 seconds (96.1%)
```

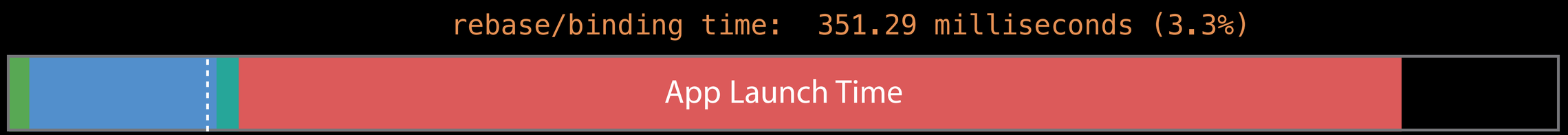
```
Total pre-main time: 10.4 seconds (100.0%)
         dylib loading time:  21.75 milliseconds (0.2%)
        rebase/binding time:  351.29 milliseconds (3.3%)
           ObjC setup time:  11.83 milliseconds (0.1%)
          initializer time:  10 seconds (94.3%)
          slowest intializers :
              MyAwesomeApp :  10.0 seconds (96.1%)
```

Rebase/Binding



Rebase/Binding

Reduce __DATA pointers



Rebase/Binding

Reduce __DATA pointers

Reduce Objective C metadata

- Classes, selectors, and categories

```
rebase/binding time:  351.29 milliseconds (3.3%)
```

Rebase/Binding

Reduce __DATA pointers

Reduce Objective C metadata

- Classes, selectors, and categories
- Reduce C++ virtual

```
rebase/binding time:  351.29 milliseconds (3.3%)
```

Rebase/Binding

Reduce __DATA pointers

Reduce Objective C metadata

- Classes, selectors, and categories

Reduce C++ virtual

Use Swift structs

```
rebase/binding time:  351.29 milliseconds (3.3%)
```

Rebase/Binding

Reduce __DATA pointers

Reduce Objective C metadata

- Classes, selectors, and categories

Reduce C++ virtual

Use Swift structs

Examine machine generated code

- Use offsets instead of pointers
- Mark read only

```
rebase/binding time:  351.29 milliseconds (3.3%)
```

Rebase/Binding

Reduce __DATA pointers

Reduce Objective C metadata

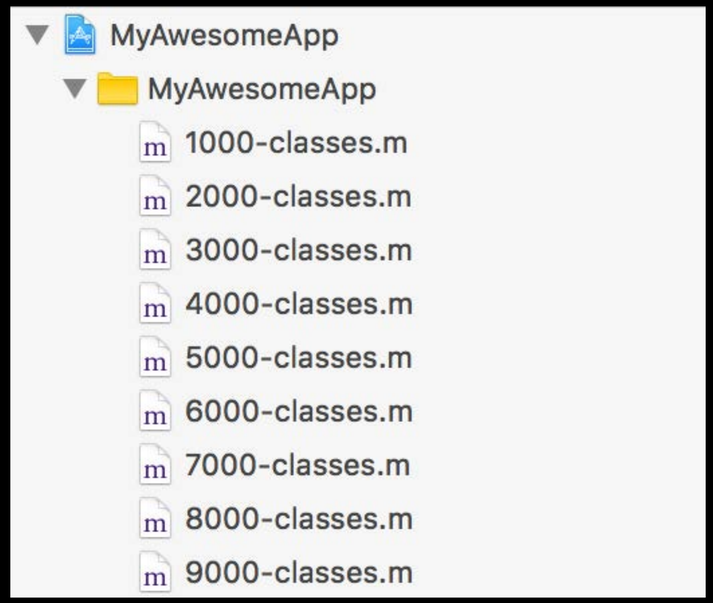
- Classes, selectors, and categories

Reduce C++ virtual

Use Swift structs

Examine machine generated code

- Use offsets instead of pointers
- Mark read only




```
rebase/binding time:  351.29 milliseconds (3.3%)
```

Rebase/Binding

Reduce __DATA pointers

Reduce Objective C metadata

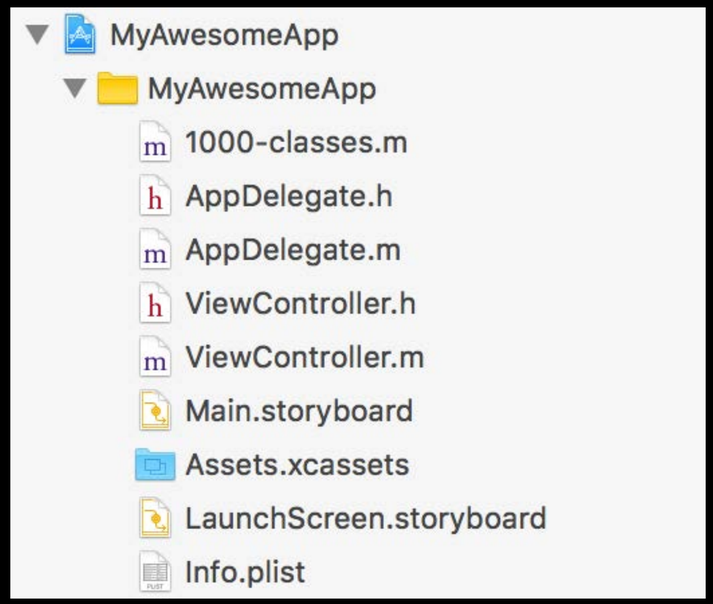
- Classes, selectors, and categories

Reduce C++ virtual

Use Swift structs

Examine machine generated code

- Use offsets instead of pointers
- Mark read only




```
rebase/binding time:  19.33 milliseconds (0.2%)
```

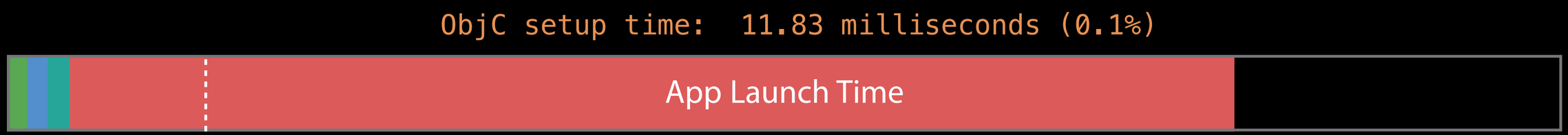
```
Total pre-main time: 10.1 seconds (100.0%)
         dylib loading time:  21.75 milliseconds (0.2%)
        rebase/binding time:  19.33 milliseconds (0.2%)
            ObjC setup time:  11.83 milliseconds (0.1%)
           initializer time:  10 seconds (99.4%)
          slowest intializers :
                MyAwesomeApp :  10.0 seconds (99.3%)
```

```
Total pre-main time: 10.1 seconds (100.0%)
         dylib loading time:  21.75 milliseconds (0.2%)
        rebase/binding time:  19.33 milliseconds (0.2%)
            ObjC setup time:  11.83 milliseconds (0.1%)
           initializer time:  10 seconds (99.4%)
          slowest intializers :
                MyAwesomeApp :  10.0 seconds (99.3%)
```

```
Total pre-main time: 10.1 seconds (100.0%)
         dylib loading time:  21.75 milliseconds (0.2%)
        rebase/binding time:  19.33 milliseconds (0.2%)
            ObjC setup time:  11.83 milliseconds (0.1%)
           initializer time:  10 seconds (99.4%)
          slowest intializers :
                MyAwesomeApp :  10.0 seconds (99.3%)
```

ObjC Setup

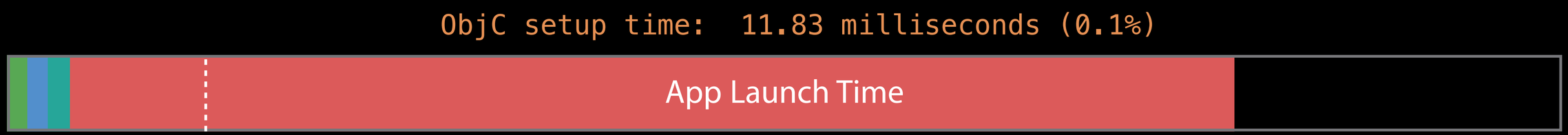
Class registration



ObjC Setup

Class registration

Non-fragile ivars offsets updated

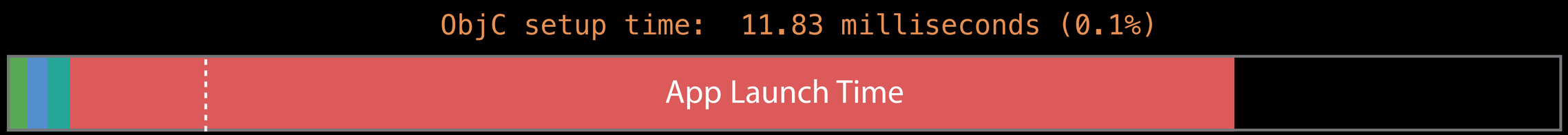


ObjC Setup

Class registration

Non-fragile ivars offsets updated

Category registration



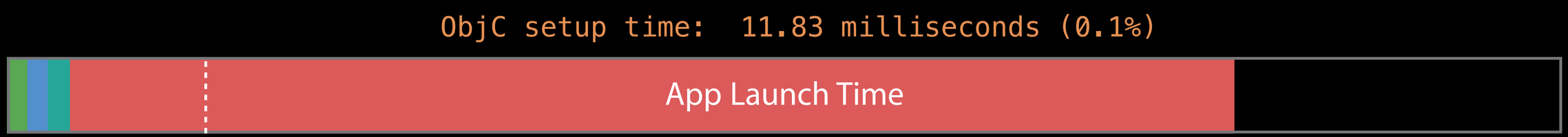
ObjC Setup

Class registration

Non-fragile ivars offsets updated

Category registration

Selector uniquing



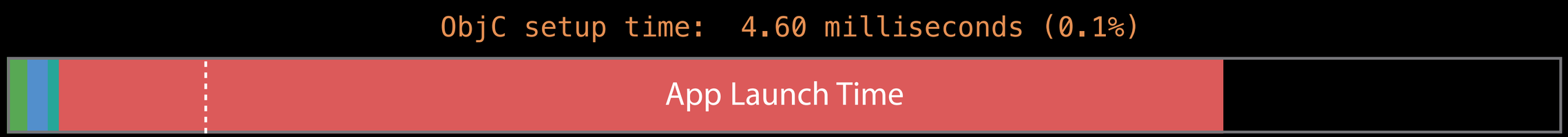
ObjC Setup

Class registration

Non-fragile ivars offsets updated

Category registration

Selector uniquing

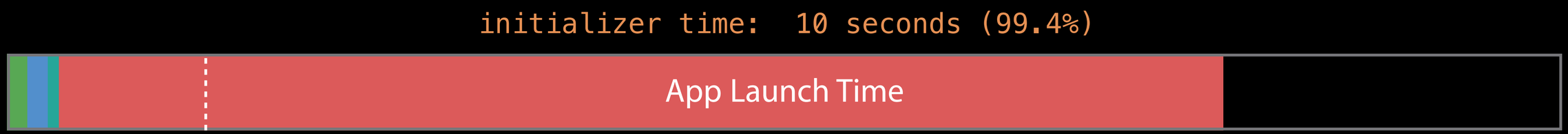


```
Total pre-main time: 10.6 seconds (100.0%)
         dylib loading time:  21.75 milliseconds (2.2%)
        rebase/binding time:  19.33 milliseconds (3.3%)
            ObjC setup time:  4.60 milliseconds (0.1%)
           initializer time:  10 seconds (94.3%)
           slowest intializers :
             MyAwesomeApp :  10.0 seconds (99.3%)
```

```
Total pre-main time: 10.6 seconds (100.0%)
         dylib loading time:  21.75 milliseconds (2.2%)
        rebase/binding time:  19.33 milliseconds (3.3%)
            ObjC setup time:  4.60 milliseconds (0.1%)
           initializer time:  10 seconds (99.4%)
           slowest intializers :
             MyAwesomeApp :  10.0 seconds (99.3%)
```

Initializers

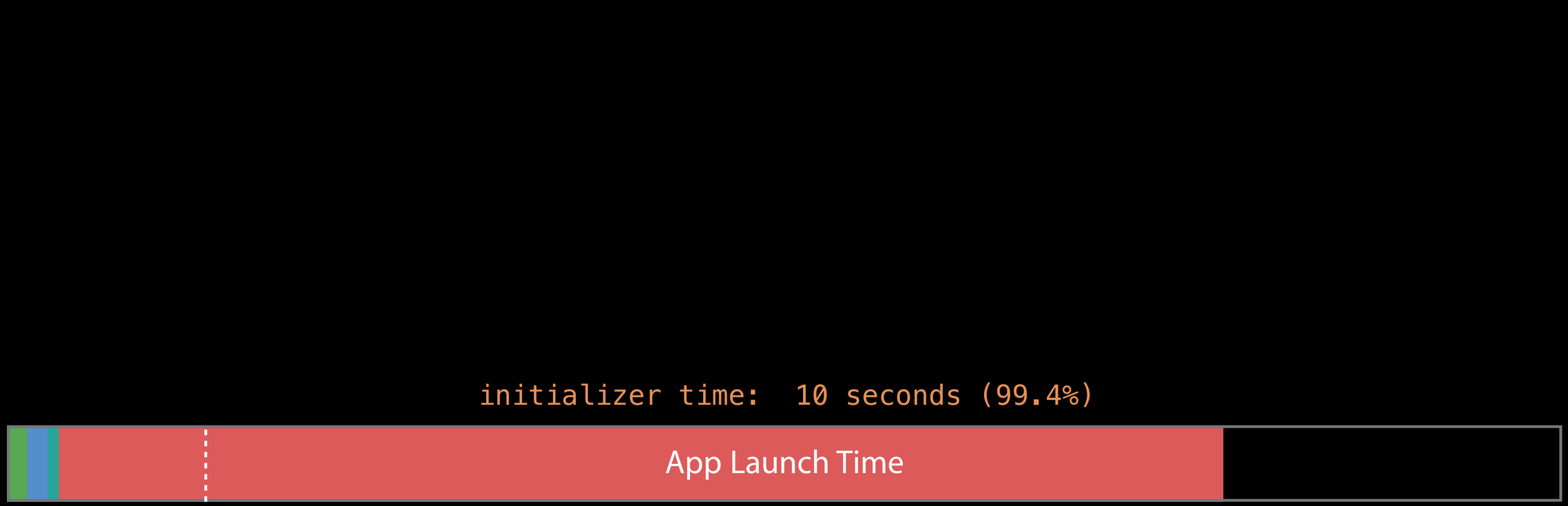
Explicit



Initializers

Explicit

ObjC `+load` methods



Initializers

Explicit

ObjC `+load` methods

- Replace with `+initiailize`

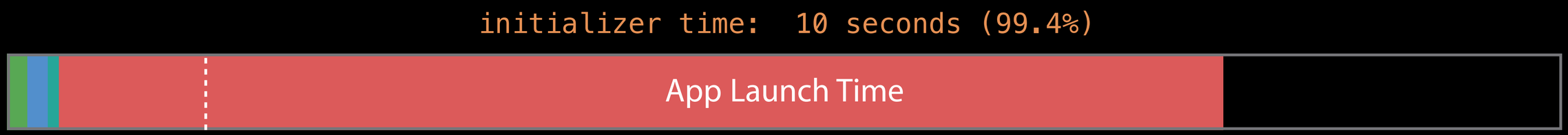
Initializers

Explicit

ObjC `+load` methods

- Replace with `+initiailize`

C/C++ __attribute__((constructor))



Initializers

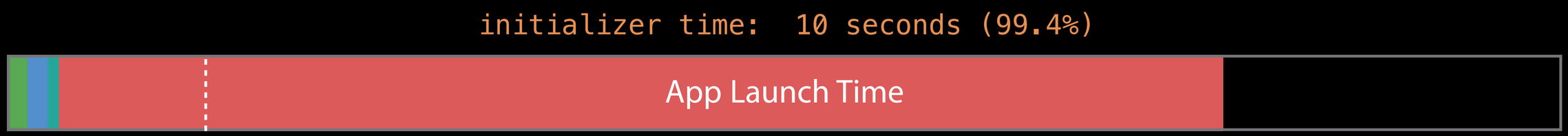
Explicit

ObjC `+load` methods

- Replace with `+initiailize`

C/C++ __attribute__((constructor))

Replace with call site initializers



Initializers

Explicit

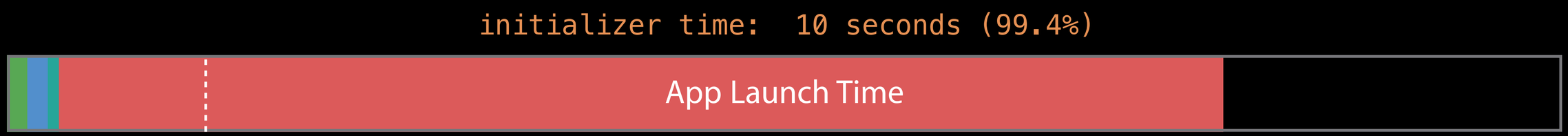
ObjC `+load` methods

- Replace with `+initiailize`

C/C++ __attribute__((constructor))

Replace with call site initializers

- `dispatch_once()`



Initializers

Explicit

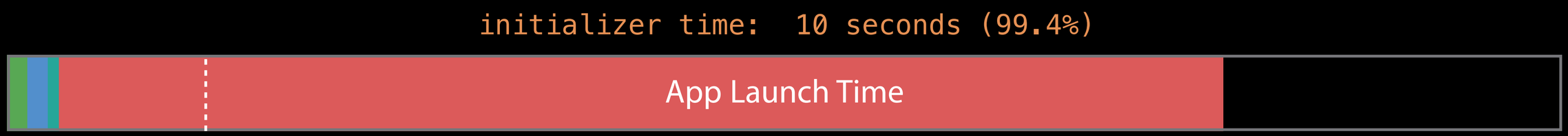
ObjC `+load` methods

- Replace with `+initiailize`

C/C++ __attribute__((constructor))

Replace with call site initializers

- `dispatch_once()`
- `pthread_once()`



Initializers

Explicit

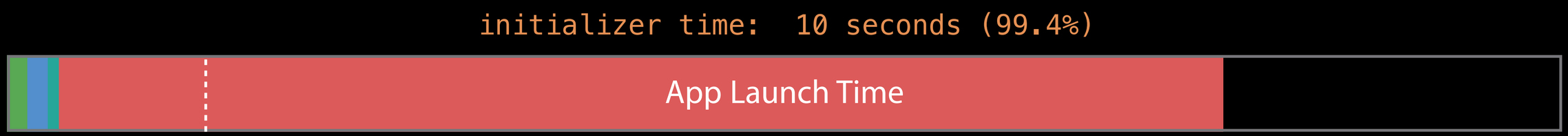
ObjC `+load` methods

- Replace with `+initiailize`

C/C++ __attribute__((constructor))

Replace with call site initializers

- `dispatch_once()`
- `pthread_once()`
- `std::once()`



Initializers

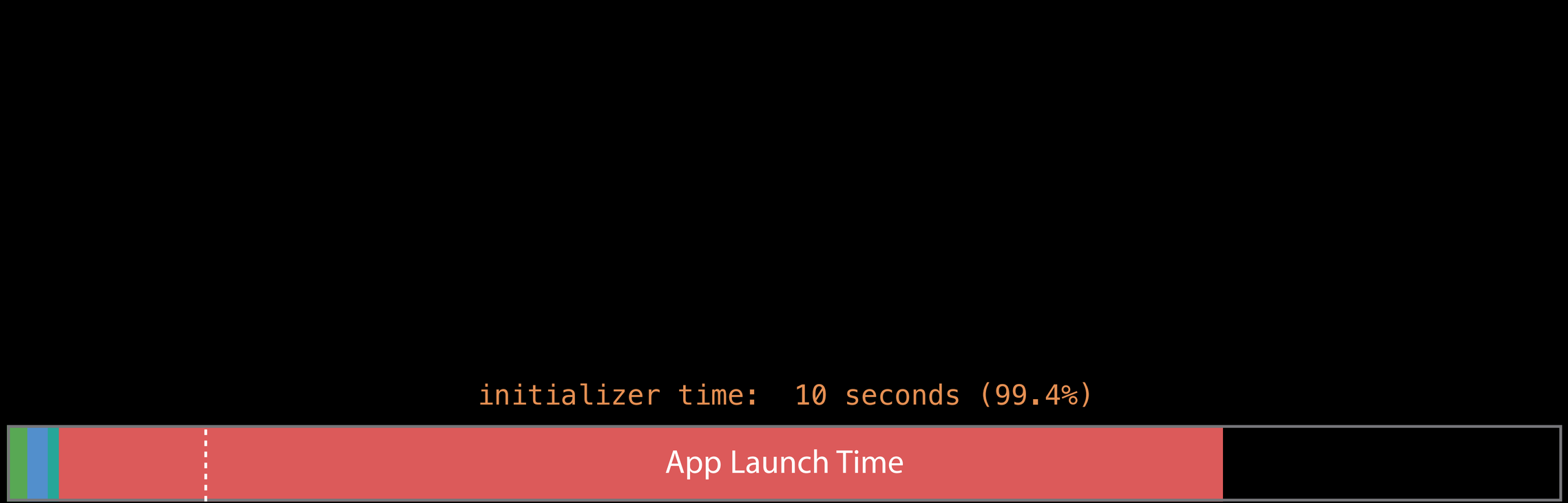
Implicit

initializer time: 10 seconds (99.4%)

Initializers

Implicit

C++ statics with non-trivial constructors

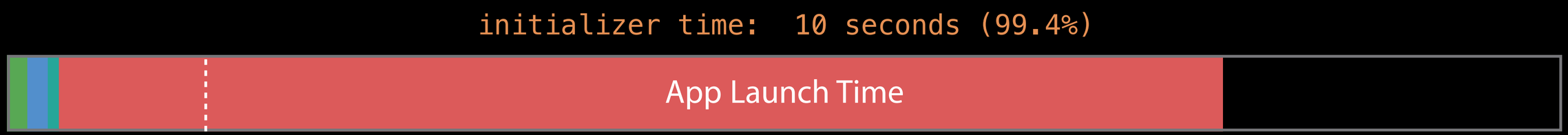


Initializers

Implicit

C++ statics with non-trivial constructors

- Replace with call site initializers

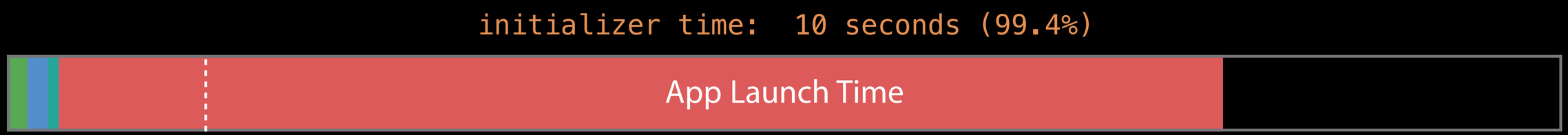


Initializers

Implicit

C++ statics with non-trivial constructors

- Replace with call site initializers
- Only set simple values (PODs)

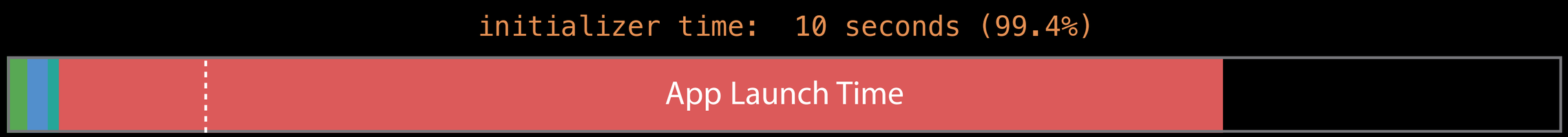


Initializers

Implicit

C++ statics with non-trivial constructors

- Replace with call site initializers
- Only set simple values (PODs)
- -Wglobal-constructors

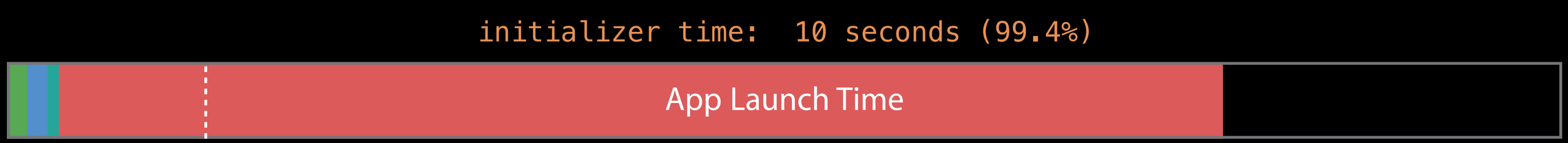


Initializers

Implicit

C++ statics with non-trivial constructors

- Replace with call site initializers
- Only set simple values (PODs)
- -Wglobal-constructors
- Rewrite in Swift



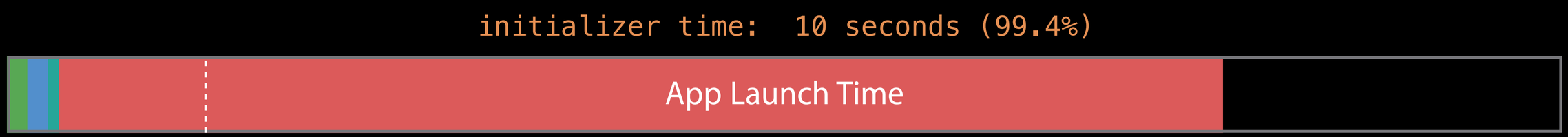
Initializers

Implicit

C++ statics with non-trivial constructors

- Replace with call site initializers
- Only set simple values (PODs)
- -Wglobal-constructors
- Rewrite in Swift

Do not call `dlopen()` in initializers



Initializers

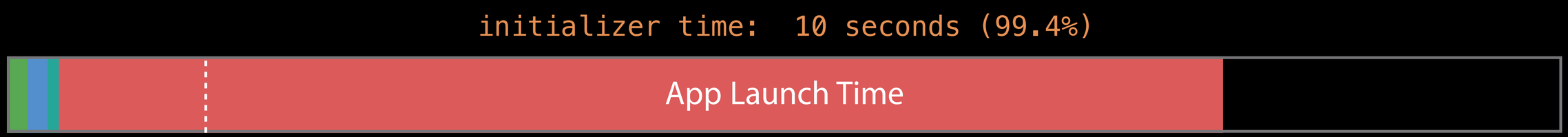
Implicit

C++ statics with non-trivial constructors

- Replace with call site initializers
- Only set simple values (PODs)
- -Wglobal-constructors
- Rewrite in Swift

Do not call `dlopen()` in initializers

Do not create threads in initializers



Initializers

Implicit

C++ statics with non-trivial constructors

- Replace with call site initializers
- Only set simple values (PODs)
- -Wglobal-constructors
- Rewrite in Swift

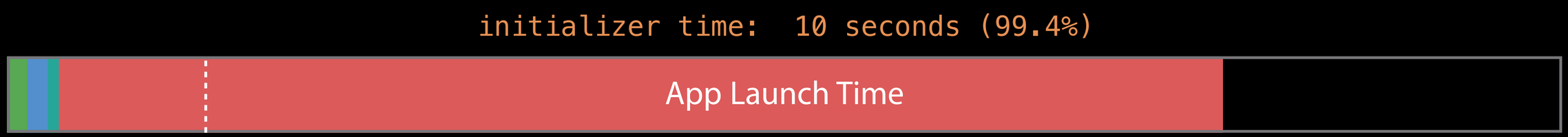
Do not call `dlopen()` in initializers

Do not create threads in initializers

```
#import <UIKit/UIKit.h>
#import "AppDelegate.h"

struct Pause {
    Pause(uint32_t i) {
        sleep(i);
    }
};

Pause onLaunch(10);
```



Initializers

Implicit

C++ statics with non-trivial constructors

- Replace with call site initializers
- Only set simple values (PODs)
- -Wglobal-constructors
- Rewrite in Swift

Do not call `dlopen()` in initializers

Do not create threads in initializers

```
#import <UIKit/UIKit.h>
#import "AppDelegate.h"

struct Pause {
    Pause(uint32_t i) {
        sleep(i);
    }
};

//Pause onLaunch(10);
```

```
initializer time:  3.96 milliseconds (7.9%)
```

Initializers

Implicit

C++ statics with non-trivial constructors

- Replace with call site initializers
- Only set simple values (PODs)
- -Wglobal-constructors
- Rewrite in Swift

Do not call `dlopen()` in initializers

Do not create threads in initializers

```
#import <UIKit/UIKit.h>
#import "AppDelegate.h"

struct Pause {
    Pause(uint32_t i) {
        sleep(i);
    }
};

//Pause onLaunch(10);
```

```
initializer time:  3.96 milliseconds (7.9%)
```

```
Total pre-main time: 49.83 milliseconds (100.0%)
         dylib loading time:  21.75 milliseconds (43.6%)
        rebase/binding time:  19.33 milliseconds (38.7%)
           ObjC setup time:  4.60 milliseconds (9.2%)
          initializer time:  3.96 milliseconds (7.9%)
     slowest intializers :
       libSystem.B.dylib : 2.80 milliseconds (5.6%)
```

```
Total pre-main time: 49.83 milliseconds (100.0%)
         dylib loading time:  21.75 milliseconds (43.6%)
        rebase/binding time:  19.33 milliseconds (38.7%)
            ObjC setup time:  4.60 milliseconds (9.2%)
           initializer time:  3.96 milliseconds (7.9%)
```

```
slowest intializers :
  libSystem.B.dylib : 2.80 milliseconds (5.6%)
```

TL;DR

Measure launch times with `DYLD_PRINT_STATISTICS`

TL;DR

Measure launch times with `DYLD_PRINT_STATISTICS`

Reduce launch times by

TL;DR

Measure launch times with `DYLD_PRINT_STATISTICS`

Reduce launch times by

- Embedding fewer dylibs

TL;DR

Measure launch times with `DYLD_PRINT_STATISTICS`

Reduce launch times by

- Embedding fewer dylibs
- Consolidating Objective-C classes

TL;DR

Measure launch times with `DYLD_PRINT_STATISTICS`

Reduce launch times by

- Embedding fewer dylibs
- Consolidating Objective-C classes
- Eliminating static initializers

TL;DR

Measure launch times with `DYLD_PRINT_STATISTICS`

Reduce launch times by

- Embedding fewer dylibs
- Consolidating Objective-C classes
- Eliminating static initializers

Use more Swift

TL;DR

Measure launch times with `DYLD_PRINT_STATISTICS`

Reduce launch times by

- Embedding fewer dylibs
- Consolidating Objective-C classes
- Eliminating static initializers

Use more Swift

`dlopen()` is discouraged

TL;DR

Measure launch times with `DYLD_PRINT_STATISTICS`

Reduce launch times by

- Embedding fewer dylibs
- Consolidating Objective-C classes
- Eliminating static initializers

Use more Swift

`dlopen()` is discouraged

- Subtle performance and deadlock issues

More Information

https://developer.apple.com/wwdc16/406

Related Sessions

Optimizing I/O for Performance and Battery Life	Nob Hill	Friday 11:00AM
Using Time Profiler in Instruments	Nob Hill	Friday 3:00PM
iOS App Performance Responsiveness	WWDC 2012	

Labs

Compiler, Objective-C, and C++ Lab	Developer Tools Lab B	Wednesday 12:00PM
Compiler, Objective-C, and C++ Lab	Developer Tools Lab B	Wednesday 1:30PM
Compiler, Optimizing App Startup Time Lab	Developer Tools Lab B	Thursday 1:30PM

WWDC16