SHARING C++ OBJECTS IN LINUX

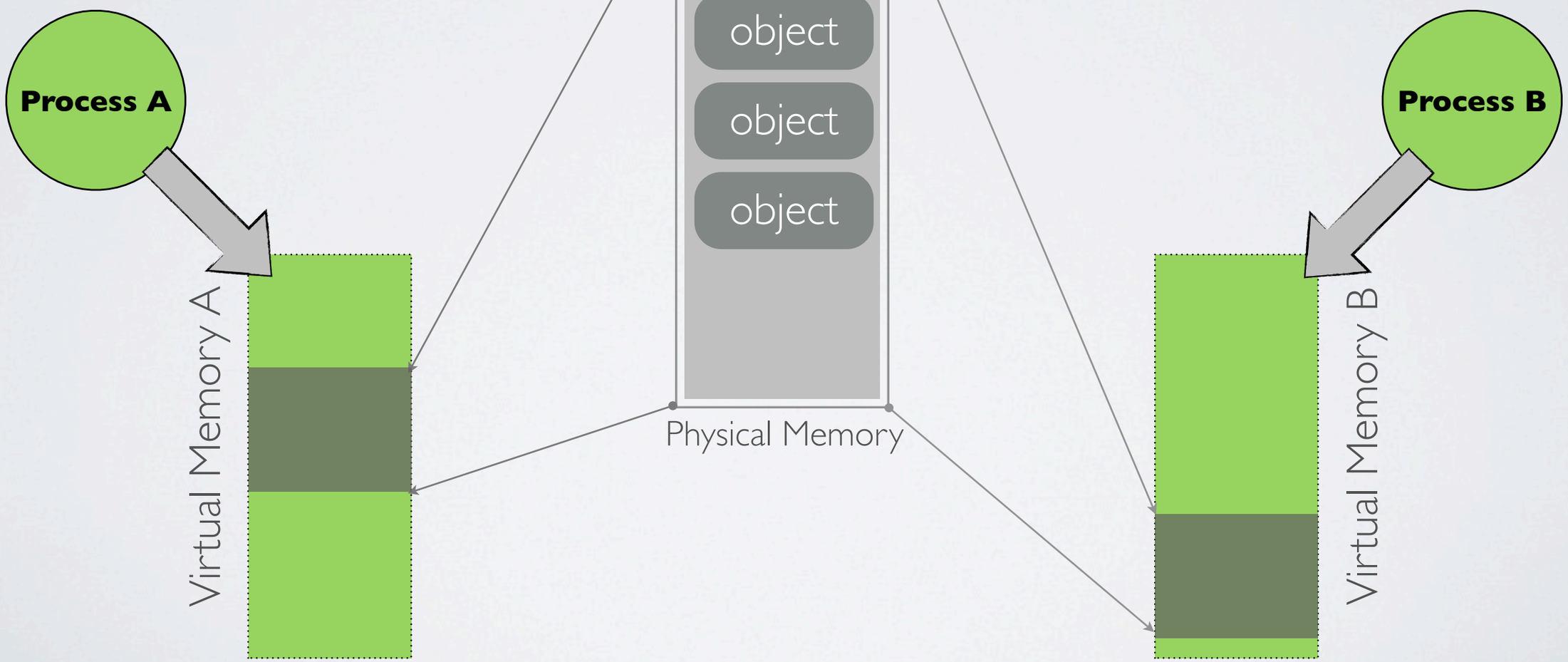
Roberto A. Vitillo (LBNL)

November 2011, Concurrency Workshop, Chicago

EXAMPLE USE CASE

Database of objects in shared memory

(simplified view...)



OBJECT SHARING

- Objects can be allocated in shared memory segments (e.g. custom allocators)
- Assumptions:
  - GCC ≥ 3 (Itanium C++ ABI)
  - Linux ≥ 2.6.12 (Address Space Layout Randomization)
  - Changing the default visibility of symbols, with visibility pragmas or linker scripts for example, is not considered
  - Binding references to global symbols to the definition within a shared library (e.g. ld's -Bsymbolic option) is not considered
- Issues:
  - Static data
  - Pointers and references
  - Virtual functions
  - Virtual inheritance

STATIC DATA

- Static data members have external linkage; each process has its own copy
- Non-const static data members reside in the object file's .data section: changes are **not** shared between the processes
- Const static members reside in the object's file .rodata section and their semantic is not altered
- Same argument applies to local static variables

POINTERS AND REFERENCES

- Pointers and references can be shared only when pointing to objects within shared segments that are mapped at the same virtual base address in all processes
  - allocate needed shared segments before forking
  - allocate shared segments at the same virtual base address in all processes
  - use “slow” offset pointers

VIRTUAL TABLES

- A class that has virtual methods or virtual bases has a set of associated virtual tables

- The tables reside in the .rodata section

- An object of such class contains pointers to the above mentioned tables (through itself or its contained base objects)

VIRTUAL TABLES

```
class A{
public:
    virtual void foo();

private:
    int m_a;
};

class B{
public:
    virtual void bar();

private:
    int m_b;
};

class C: public virtual A, public virtual B{
public:
    virtual void foo();
    virtual void car();
};
```

Vtable A	
0	offset_to_top(0)
8	A::rtti
16	A::foo()

Vtable B	
0	offset_to_top(0)
8	B::rtti
16	B::bar()

Vtable C	
0	offset_to_B(24)
8	offset_to_A(8)
16	offset_to_top(0)
24	C::rtti
32	C::foo()
40	C::car()
48	vcall_offset_foo(-8)
56	offset_to_top(-8)
64	C::rtti
72	C::foo() thunk
80	vcall_offset_bar(0)
88	offset_to_top(-18)
96	C::rtti
104	B::bar

VIRTUAL TABLES

- Virtual Tables reside in the object's file .rodata section
- Methods and thunks in the .text section
- To be able to share objects that contain pointers to vtables, that in turn contain pointers to methods, thunks and typeinfos, the vtables, methods, thunks and typeinfos need to have the same virtual addresses on all processes
- Methods can in turn reference functions and data...
  - object files have to be mapped at the same virtual base address on all processes
- Executables that don't use shared libraries are loaded at a fixed address and their initial virtual memory layout is the "same" even for unrelated processes
  - if -fPIE is not enabled

SHARED LIBRARIES

- We know that the shared libraries need to be mapped at the same address in all processes
- If the shared library is dynamically linked to an executable that forks the cooperating processes, no changes are needed as the mapping is inherited

ASLR in action

```
vitillo@eniac $ ldd /usr/bin/gcc
linux-vdso.so.1 =>  (0x00007fff99e6f000)
libc.so.6 => /lib64/libc.so.6 (0x00002aefcc69d000)
libdl.so.2 => /lib64/libdl.so.2 (0x00002aefcc9f5000)
/lib64/ld-linux-x86-64.so.2 (0x00002aefcc27d000)

vitillo@eniac $ ldd /usr/bin/gcc
linux-vdso.so.1 =>  (0x00007fff1c12f000)
libc.so.6 => /lib64/libc.so.6 (0x00002aabda159000)
libdl.so.2 => /lib64/libdl.so.2 (0x00002aabda4b1000)
/lib64/ld-linux-x86-64.so.2 (0x00002aabd9d39000)
```

- For unrelated processes, that dynamically link a library, we can't rely on the load order defined in the .dynamic section
  - Address Space Layout Randomization
  - Preloading can change the load order
- Dynamic loading can be an issue for related and unrelated processes

SHARED LIBRARIES

- Workaround:
  - Use a linker script to set the first PT_LOAD segment's p_vaddr to assign to each DSO a fixed unique load address (or use prelink)
    - feasible for x86-64
- The dynamic linker can be modified to enforce the loading of the shared library at the specified address since otherwise p_vaddr is taken only as a suggestion
- One could think of more sophisticated solutions whose end effect reduces to the above mentioned one
    - E.g. set a flag in the .dynamic section of the shared libraries that contain the vtables of the classes whose objects we want to share and set p_vaddr only for those
- Similar considerations can be made for dynamically loaded shared libraries

SHARED LIBRARIES

- The One Definition Rule requires to have a single definition even if many objects in C++ are not clearly part of a single object

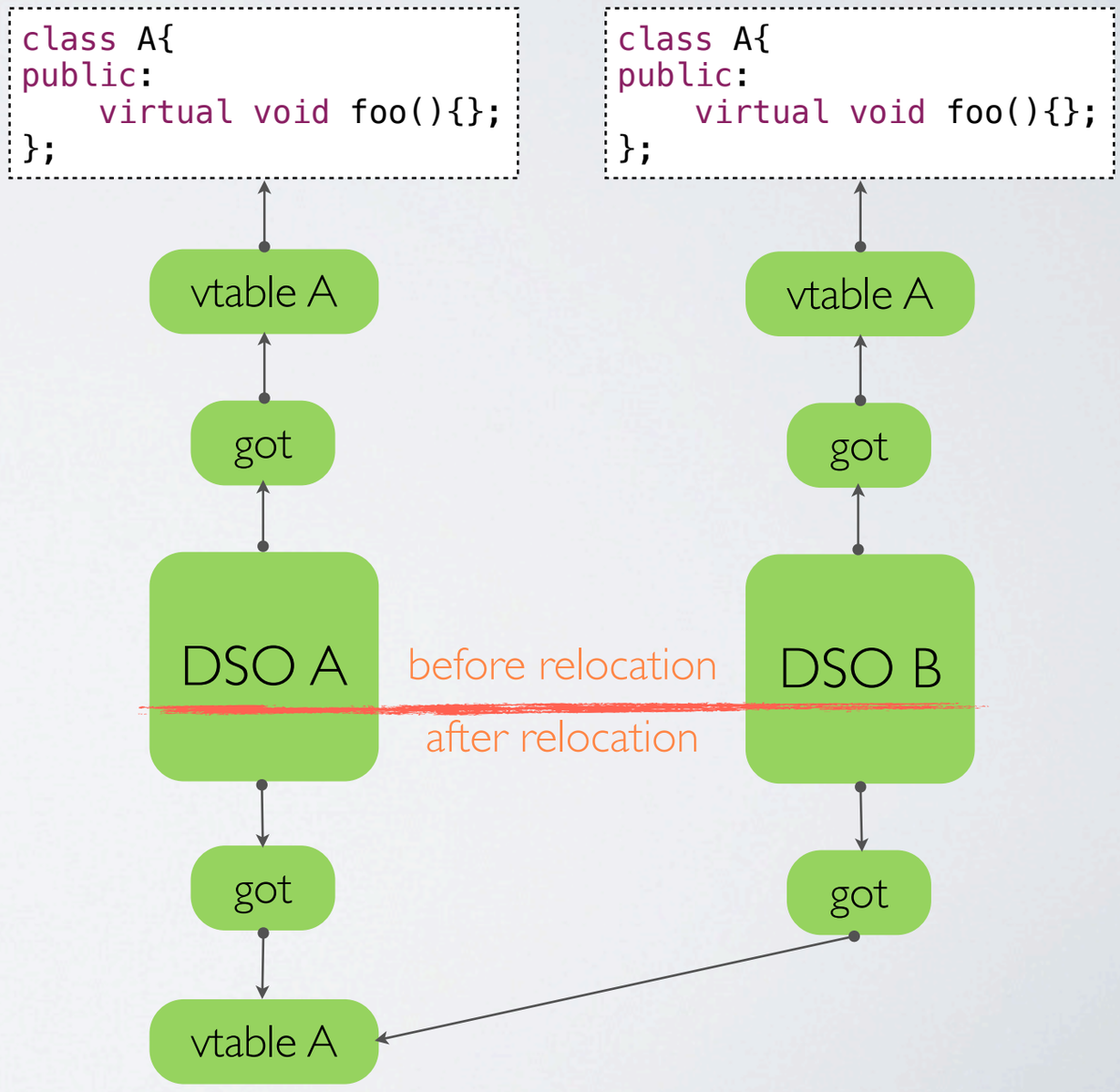
- The virtual table for a class is emitted in the same object containing the definition of its key function

- The key function is the first non-pure virtual, non-inline function at the point of class definition

- If there is no key function then the vtable is emitted **everywhere** used

SHARED LIBRARIES

- E.g. each shared library that is using a class with only inline virtual functions has a local definition of the vtable
- All shared libraries that contain a copy of the virtual table should be mapped at the same address...
- ...unless we are certain that the first DSO with the incriminated vtable is loaded at the same virtual address in all processes
  - relocation and symbol interposing take care of the rest
- Workaround: classes whose objects need to be shared are required to have a key function



VIRTUAL INHERITANCE

- Virtually inherited objects have **pointers** to address base objects

- How those pointers are implemented is not defined by the C++ standard

- On Linux the ABI requires them to be implemented as offsets

  - no issue

CONCLUSION

- A slow, costly but portable solution would require a new layer of indirection
- The proposed workarounds may be non-portable but enable processes to share objects without overheads and deep changes to the toolchain
- Proposed solution that requires a minimum amount of work:
  - sharable classes have a key function
  - an executable is dynamically linked to the libraries with the sharable vtables
  - dynamic loading of libraries with sharable vtables is only allowed before forking
  - the cooperating processes are forked

RESOURCES

- Boost Interprocess - http://www.boost.org
- C++, ISO/IEC 14882 International Standard
- System V ABI - http://www.sco.com/developers/gabi/latest/contents.html
- Itanium C++ ABI - http://sourcery.mentor.com/public/cxx-abi/abi.html
- Address Space Layout Randomization - http://en.wikipedia.org/wiki/Address_space_layout_randomization
- Prelink - http://people.redhat.com/jakub/prelink.pdf
- How To Write Shared Libraries - http://www.akkadia.org/drepper/dsohowto.pdf
- Executable and Linking Format (ELF) Specification - http://pdos.csail.mit.edu/6.828/2011/readings/elf.pdf