

# Making Address Spaces Smaller

ICS332 — Operating Systems

Henri Casanova (henric@hawaii.edu)

Spring 2018

# Smaller Address Spaces

- Having small address spaces is always a good idea
- This is good for swapping: don't swap as often (because if address spaces are small, then RAM looks bigger), not as slow to swap (because reading/writing a smaller address space from/to disk is faster)

# Smaller Address Spaces

- Having small address spaces is always a good idea
- This is good for swapping: don't swap as often (because if address spaces are small, then RAM looks bigger), not as slow to swap (because reading/writing a smaller address space from/to disk is faster)
- Three common-place techniques:
  - Dynamic Memory Allocation
    - Ask programs to tell the OS exactly how much memory they need (`malloc`, `new`) so that we don't always allocate the maximum allowed RAM to each process
    - You all know about this one
  - Dynamic Loading
    - Show of hands, who's heard of this?
  - Dynamic Linking
    - Show of hands, who's heard of this?

# Dynamic Loading

- **Dynamic loading**: only load code/text when it's needed
- Dynamic loading is the **programmer responsibility**
  - The OS is not involved, although it provides tools to make dynamic loading possible
- Supported in all (decent) programming languages / OSes:
  - in C/C++:
    - POSIX: `dlopen`, `dlsym`...
    - Windows: `LoadLibrary`
  - e.g. [in Python](#) (let's look at this one...)
    - `import` statement anywhere in the program!
  - e.g. [in Java](#)
    - `ClassLoader` class and others
- Dynamic unloading is usually possible
- Note: We talk of *dynamic loading* but not of *static loading* (the default behavior)

# Static / Dynamic Linking

- Static Linking is the historical way of reusing code
  - Add the assembly code of useful functions (printf...) collected in an **archive** or **library** to your own executable.
  - e.g., libc.a for Linux; MSVCRT.LIB for Windows)
- Example on Linux: `gcc -static HelloWorld.c -o HelloWorld`  
`nm HelloWorld; objdump -d HelloWorld`

# Static / Dynamic Linking

- Static Linking is the historical way of reusing code
  - Add the assembly code of useful functions (printf...) collected in an **archive** or **library** to your own executable.
  - e.g., libc.a for Linux; MSVCRT.LIB for Windows)
- Example on Linux: `gcc -static HelloWorld.c -o HelloWorld`  
`nm HelloWorld; objdump -d HelloWorld`
- Issue 1: Large text
  - On my Linux VM, the HelloWorld executable is 892 KiB!
- Issue 2: Some code is (very likely) duplicated in memory
  - My program is (very likely) not the only one to use printf!

# Static / Dynamic Linking

- Static Linking is the historical way of reusing code
  - Add the assembly code of useful functions (printf...) collected in an **archive** or **library** to your own executable.
  - e.g., `libc.a` for Linux; `MSVCRT.LIB` for Windows)
- Example on Linux: `gcc -static HelloWorld.c -o HelloWorld`  
`nm HelloWorld; objdump -d HelloWorld`
- Issue 1: Large text
  - On my Linux VM, the `HelloWorld` executable is 892 KiB!
- Issue 2: Some code is (very likely) duplicated in memory
  - My program is (very likely) not the only one to use `printf`!
- **Key idea:** Why not share text (i.e., code) between processes in a similar way as data can be shared through shared memory?

# Dynamic Linking

- In spirit very similar to dynamic loading
- But **the OS is in charge of loading the code**
- While also means that the OS needs to be told where to load it (at compile time)



# Dynamic Linking

- In spirit very similar to dynamic loading
- But **the OS is in charge of loading the code**
- While also means that the OS needs to be told where to load it (at compile time)
- The code is shared in **shared libraries**:
  - e.g., `libc.so` for Linux (so = shared object)
  - `MSVCRT.DLL` for Windows (DLL = Dynamic-link library)

# Dynamic Linking

- In spirit very similar to dynamic loading
- But **the OS is in charge of loading the code**
- While also means that the OS needs to be told where to load it (at compile time)
- The code is shared in **shared libraries**:
  - e.g., `libc.so` for Linux (so = shared object)
  - `MSVCRT.DLL` for Windows (DLL = Dynamic-link library)
- Linux example: `gcc -shared -fPIC HelloWorld.c -o HelloWorld`  
`nm HelloWorld; objdump -d HelloWorld`
- On my Linux VM, the HelloWorld executable is 8.5KiB (compared to the 892KiB statically linked one)!

# Shared Libraries – How does it work?

- When dynamic linking is enabled, the linker just puts a **stub** in the binary for each shared library routine reference

# Shared Libraries – How does it work?

- When dynamic linking is enabled, the linker just puts a **stub** in the binary for each shared library routine reference
- That **stub** is a piece of code that:
  - checks whether the routine is loaded in memory
  - if not, then loads it into memory “shared” (with all processes)
  - then replaces itself with a simple call to the routine
  - future calls will be “for free”

# Shared Libraries – How does it work?

- When dynamic linking is enabled, the linker just puts a **stub** in the binary for each shared library routine reference
- That **stub** is a piece of code that:
  - checks whether the routine is loaded in memory
  - if not, then loads it into memory “shared” (with all processes)
  - then replaces itself with a simple call to the routine
  - future calls will be “for free”
- Chances are that when you run HelloWorld, the `printf` code is already in memory:

# Shared Libraries – How does it work?

- When dynamic linking is enabled, the linker just puts a **stub** in the binary for each shared library routine reference
- That **stub** is a piece of code that:
  - checks whether the routine is loaded in memory
  - if not, then loads it into memory “shared” (with all processes)
  - then replaces itself with a simple call to the routine
  - future calls will be “for free”
- Chances are that when you run HelloWorld, the `printf` code is already in memory:  $\Rightarrow$  Save space!

# Shared Libraries – How does it work?

- When dynamic linking is enabled, the linker just puts a **stub** in the binary for each shared library routine reference
- That **stub** is a piece of code that:
  - checks whether the routine is loaded in memory
  - if not, then loads it into memory “shared” (with all processes)
  - then replaces itself with a simple call to the routine
  - future calls will be “for free”
- Chances are that when you run HelloWorld, the `printf` code is already in memory:  $\Rightarrow$  Save space!  $\Rightarrow$  Save time!

# Shared Libraries – Easy Updates

- Haven't you wondered how come you can update your system (i.e., libraries) and not have to recompile all your executables???
- This would be insanely inconvenient!



# Shared Libraries – Easy Updates

- Haven't you wondered how come you can update your system (i.e., libraries) and not have to recompile all your executables???
- This would be insanely inconvenient!
- Provided the APIs haven't not changed you can just:
  - Replace a shared library (.so, .dll) by a new one
  - Ask the system to “reload” it
  - And now it all magically works!
  - If the update was critical (i.e., security) then a reboot may be required

# Shared Libraries – Easy Updates

- Haven't you wondered how come you can update your system (i.e., libraries) and not have to recompile all your executables???
  - This would be insanely inconvenient!
- Provided the APIs haven't not changed you can just:
  - Replace a shared library (.so, .dll) by a new one
  - Ask the system to “reload” it
  - And now it all magically works!
  - If the update was critical (i.e., security) then a reboot may be required
- Dynamic Linking requires help from the OS
  - To break memory isolation and allow shared text segments among processes
  - We will see that this comes “for free” with virtual memory (next Module)

# Details of Shared Libraries

- On Linux system the `ldd` command will print the shared libraries required by a program
- For instance, let us look at the shared libraries used by `/bin/date`
  - The compiler adds stuff in the executable so that `ldd` can find this information and display it
- It turns out that, in Linux, you can override functions from loaded shared libraries by creating yourself a small shared library
- Let's try this to do something useful...

# Example: C and Memory Leaks

- As you know, in C you allocate/free memory with `malloc()` and `free()`
- Every call to `malloc()` should have a matching call to `free()`
- This is easier said than done, as you might know
  - But perhaps you didn't care about memory leaks when writing C

# Example: C and Memory Leaks

- As you know, in C you allocate/free memory with `malloc()` and `free()`
- Every call to `malloc()` should have a matching call to `free()`
- This is easier said than done, as you might know
  - But perhaps you didn't care about memory leaks when writing C
- Wouldn't it be great if somehow the code counted calls to `malloc()` and `free()`?

# Example: C and Memory Leaks

- As you know, in C you allocate/free memory with `malloc()` and `free()`
- Every call to `malloc()` should have a matching call to `free()`
- This is easier said than done, as you might know
  - But perhaps you didn't care about memory leaks when writing C
- Wouldn't it be great if somehow the code counted calls to `malloc()` and `free()`?
- Let's do that with a small shared library...

# First Step: Overriding `exit()`

- I've written the code for a shared library that overrides the `exit()` library function
- To create a shared library `.so` file:
  - `gcc -fPIC -DPIC -c custom_shared_library.c -o custom_shared_library.o`
  - `ld -shared -ldl -o custom_shared_library.so custom_shared_library.o`
- Then, one can *enable* the shared library by setting the `LD_PRELOAD` environment variable to the path of the shared library
- I've put together the code above and a Makefile that does everything, including compiling a small program that just does an `exit` (See the "Example source code" reading in this module)
- Let's look at all this and run it...

# Overriding malloc() and exit()!

- Let's now augment our custom shared library to override `malloc()` and `free()`
- The code is on the web site (in the "Example source code" reading), but let's try to do it live...
  - Objective: keep counts of calls to `malloc()` and calls to `free()`, and print a warning if they don't match!
- Let's then try to run a leaky C program, a Java program, make, etc.
- Note we don't have to re-compile any of those programs!



# This is very useful!

- We could augment what we just did to make it way more useful and user friendly
- For instance, we could find out where calls to `malloc()` are placed (i.e., which lines of code) and then report on which ones were not freed!
- Turns out, tools exist that do this already: `valgrind`, `purify`...

# This is very useful!

- We could augment what we just did to make it way more useful and user friendly
- For instance, we could find out where calls to `malloc()` are placed (i.e., which lines of code) and then report on which ones were not freed!
- Turns out, tools exist that do this already: `valgrind`, `purify`...
  - Let's run `valgrind` on our leaky C program...
- More generally: You're not happy with the performance of one function in a standard lib? Rewrite it and replace it on-the-fly!

# This is very useful!

- We could augment what we just did to make it way more useful and user friendly
- For instance, we could find out where calls to `malloc()` are placed (i.e., which lines of code) and then report on which ones were not freed!
- Turns out, tools exist that do this already: `valgrind`, `purify`...
  - Let's run `valgrind` on our leaky C program...
- More generally: You're not happy with the performance of one function in a standard lib? Rewrite it and replace it on-the-fly!
- And since this is very useful / powerful, it can also be dangerous
  - If you have a bug in your shared library, then you're stuck and nothing will work (e.g., "you broke `malloc()`!!")
  - Which is why we test with the `LD_PRELOAD` environment variable instead of making the new shared library the default system-wide

# Conclusions

- In the previous set of lecture notes we saw that we may have to swap because we run out of RAM
- Making address spaces as small as possible is thus a good idea
  - Won't have to swap as much
  - Not as costly to swap when swapping is needed
- **Bottom Line:** do not waste bytes!
- Part of this is on the developer:
  - Just use space-efficient data structures
  - Use *Dynamic Memory Allocation*
- Part of this is provided by languages/compilers/OS:
  - Dynamic loading
  - Dynamic linking
- We'll have a QUIZ on this entire module next week