Compiler in build process

- Translates C code in assembly code.
- Example of GCC call.

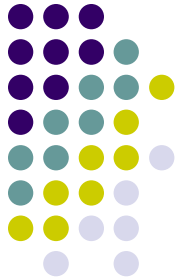
```
gcc –S file_name.c –o file_name.s
```

- Command line switch -S tells compiler to generate file with the assembly code, otherwise object file will be generated directly.
- GCC does not generate object file itself, it calls the assembler.

```
int main()
{
   printf ("Hello World!\n");
}
```

```
        .file   1 "hello.c"
        .section .mdebug.abi32
        .previous
        .rdata
        .align  2
.LC0:
        .ascii  "Hello World!\000"
        .text
        .align  2
        .globl  main
        .set    nomips16
        .ent    main
        .type   main, @function
main:
        .frame  $sp,0,$31    # vars= 0, regs= 0/0, args= 0, gp= 0
        .mask   0x00000000,0
        .fmask  0x00000000,0
        la      $4,.LC0
        j       puts
        .end    main
        .size   main, .-main
        .ident  "GCC: (GNU) 4.5.0"
```

Assembler in build process



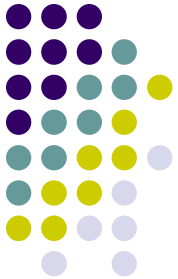
- Translates assembly file in the object file.
- Calling assembler from command line:
  can be done directly, but it can also be done through GCC

```
as file_name.s –o file_name.o
gcc file_name.s –o file_name.o
```

- Object file contains the following information in binary format:
  - Machine code
  - Initialized data
  - Symbol table
  - Relocation information
  - Debug information
    - connection between C identifiers and physical resources
    - connection between C lines and instructions

- With objdump tool (or something similar) it is possible to list the content of an object file in textual format.

Object file



- Objdump -t

```
SYMBOL TABLE:
l    df *ABS*        00000000 link.c
l    d  .text        00000000 .text
l    d  .data        00000000 .data
l    d  .bss         00000000 .bss
l    d  .comment     00000000 .comment
g     F .text        0000001d foo
      *UND*          00000000 func
      *UND*          00000000 var
```

- Objdump -d

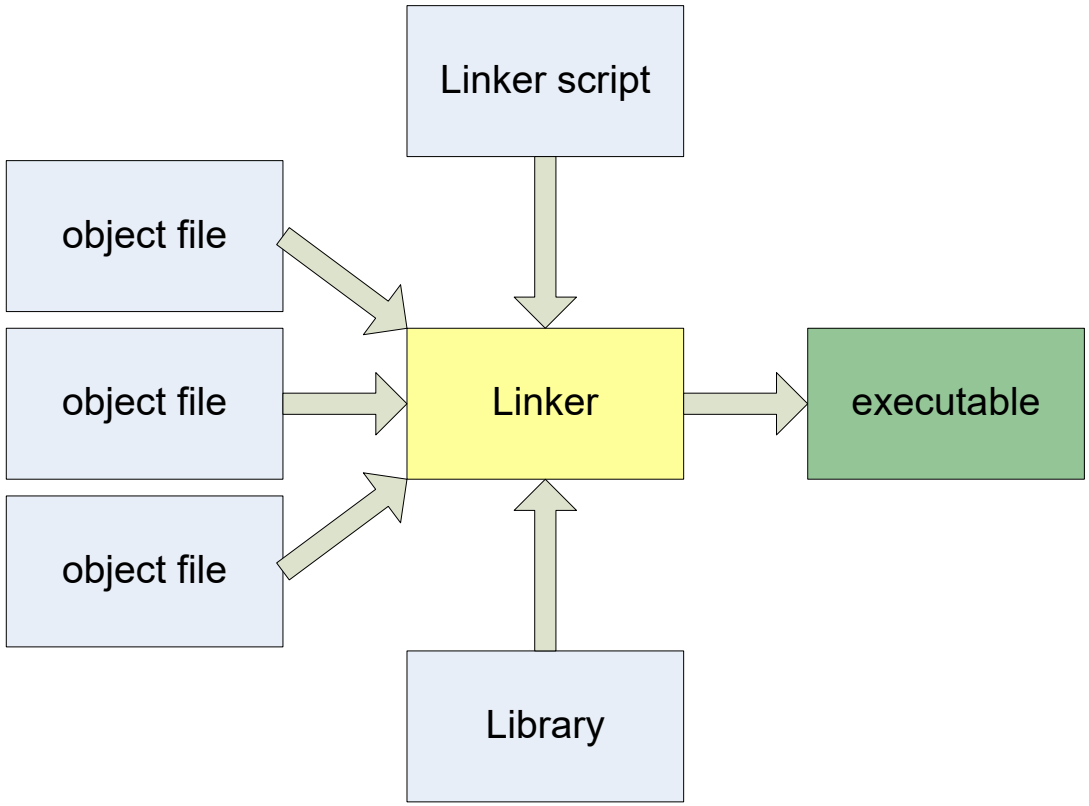
```
00000000 <foo>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 18                sub    $0x18,%esp
   6:   8b 45 08                mov    0x8(%ebp),%eax
   9:   89 04 24                mov    %eax,(%esp)
   c:   e8 fc ff ff ff          call   d <foo+0xd>
  11:   a3 00 00 00 00          mov    %eax,0x0
  16:   a1 00 00 00 00          mov    0x0,%eax
  1b:   c9                      leave
  1c:   c3                      ret
```

- Objdump -h

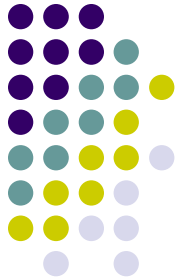
```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000001d  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000054  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000054  2**2
                  ALLOC
  3 .comment      00000024  00000000  00000000  00000054  2**0
                  CONTENTS, READONLY
  4 .note.GNU-stack 00000000  00000000  00000000  00000078  2**0
                  CONTENTS, READONLY
```

Linker 1/2

- The last phase in building process
- Links several object files into single executable file



Linker 2/2



- Linker's main task is to resolve interdependencies of the object files:
  - Calling extern functions
  - Accessing extern variables

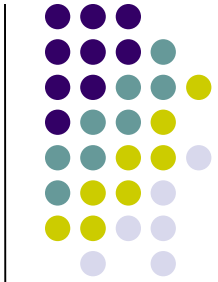
```
extern int var;
extern int func(int);

int foo(int a)
{
    var = func(a);
    return var;
}
```

```
foo:
    addiu   $sp,$sp,-24
    sw      $31,20($sp)
    jal     func
    lw      $31,20($sp)
    sw      $2,var
    j       $31
    addiu   $sp,$sp,24
```

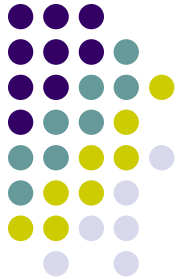
- Symbols in red have to be resolved during linking process.

Loader

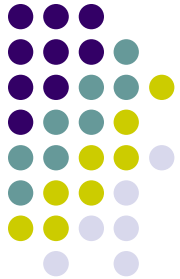


- Loader is very often a part of the operating system. Even when it exist as a separate tool (whether on host or target platform), it is still tightly related to operating system.
- Loading includes physical transfer of machine instructions from external memory to processor's address space.
- Beside machine instructions, processor memory has to be allocated (during loading) for static storage duration variables. Initial values for those variables are already stored in executable file. Those values must also be transferred to the operating memory.
- For uninitialized static storage duration variables it is not necessary to transfer values, only to allocate memory for them in special memory segment. That segment is called **bss** segment and everything in it set initially to 0.

Loader – local variables and dynamically allocated memory



- Local variables (automatic storage duration) and dynamically located objects (allocated storage duration) are not handled by loader:
  - **Local variables** are usually located in registers or program stack. Therefore, all addresses are relative to stack pointer, and that is dynamic category.
  - **Dynamically allocated objects** are allocated during run time, so loader is unaware of them.



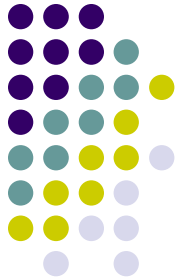
Linking kinds

- There are three ways to link:

1) Static linkage (during linking)
   - Instructions of every extern function ends up inside the executable file.
   - Function calls are made directly, since address is resolved during linkage.

2) Dynamic linking at load time
   - Extern functions are located in some separate space.
   - Operating system maps addresses of extern functions to address space of the running program.
   - Calls are made through Procedure Linkage Table (PLT), which is partly generated during linking but is completed during loading (every extern symbol has its entry in PLT)

3) Dynamic linking at run time
   - There are no real extern functions.
   - The program contains explicit code which maps function addresses to its own address space.
   - In other words, pointers to functions are being explicitly retrieved in run time.



Static library

- Example:

avglib.h:

```
double avg(double a, double b);
```

avglib.c:

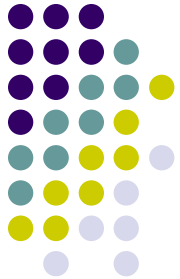
```
double avg(double a, double b)
{
    return (a + b) / 2.0;
}
```

- Building the library:

```
gcc -c avglib.c
ar rs libavg.a avglib.o
```

- The first line generates object file.
- The second line calls achiever tool (**ar**), which is used for creating library.
- Convention says that library names should start with “lib”, and that extension should be “.a”
- Static library is, in fact, just a collection of object files.

Using static library



- Example:

app.c:

```
#include "avglib.h"

int main()
{
    printf("average value %lf\n",
    avg(3.7, 4.6);
    return 0;
}
```

- Building program:

```
gcc --static -I../include -L../lib -o app app.c -lavg
```

- --static tells linker (which is called by GCC) to use static version of the library (without the switch, dynamic version of the library will be used – that is by default)
- -lavg tells linker to link the program with library “libavg” (the beginning of library's name is assumed to be “lib”)

Dynamic (shared) library

- Example:

avglib.h:

```
double avg(double a, double b);
```

avglib.c:

```
double avg(double a, double b)
{
    return (a + b) / 2.0;
}
```

- Building the library:

```
gcc -c –fpic avglib.c
gcc –shared -o libavg.so avglib.o
```

- The first line generates object library, but with switc **–fpic**. That switch tells compiler to generate position independent code. This is important because the generated code should be able to work on any address. Also, this switch has a meaning only on some platform (on some other platforms, code is always position independent)
- In the second line, switch **–shared** tells compiler to generate dynamic, i.e. shared library, from the listed object files.
- By convention, library name should start with “lib”, and its extension has to be “.so”.

Using dynamic (shared) library

- Example:

app.c:

```
#include "avglib.h"

int main()
{
    printf("average value %lf\n",
    avg(3.7, 4.6));
    return 0;
}
```

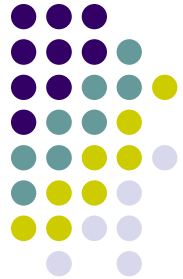
- Building program:

```
gcc -I../include -L../lib -o app app.c -lavg
```

- -lavg tell linker to link program with libavg.so library (again, we see that the beginning of library's name is expected to be "lib")
- Dynamic (shared) library is the default type of library.
- Library has to be on the same path as the executable file. If not, path to the library has to be specified in the following way:

```
export LD_LIBRARY_PATH=/library/path:$LD_LIBRARY_PATH
```

Dynamic linking in runtime



- The same dynamic (shared) library is used.
- Example:

```
#include <dlfcn.h>
#include "avglib.h“

int main()
{
    void* handle;
    double (*avg)(double,double);
    char* error;

    handle = dlopen("libavg.so", RTLD_LAZY);
    if (handle == NULL)
    {
       fputs(dlerror(), stderr);
       exit(1);
    }

    avg = dlsym(handle, "avg");
    if ((error = dlerror()) != NULL)
    {
       fputs(error, stderr);
       exit(1);
    }

    printf("%f\n", (*avg)(3.7, 4.6));
    dlclose(handle);
    return 0;
}
```

- There has to be a code that will:
  - Open dynamic library

```
void* dlopen(const char* filename, int flag);
```

  - Retrieve symbol addresses (variables or functions) that will be used

```
void* dlsym(void* handle, const char* symbol);
```

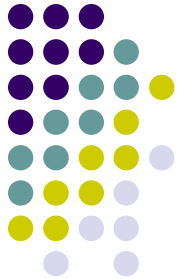
  - Close dynamic library

```
int dlclose(void* handle);
```

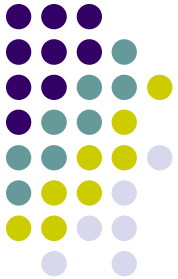
- The library is never mentioned during program build

gcc -I../include -L../lib -o app app.c

Which linkage to use?



- Advantages of static linking:
  - Function calls are faster (no indirection)
  - Program starting time is shorted, because no time is spent on linking
  - Everything is in one executable file – program distribution is simpler
- Advantages of dynamic linking at load time:
  - Smaller executable file
  - There is no multiplication of code if several different programs use the same library (there is only one instance of the library, and the programs use it – “shared”)
  - New library versions (improved in many ways) can be used by the program, without re-building it (as long as the interface is not changed)
- Advantages of dynamic linking at run time:
  - Time needed for linking (and loading the library) will be spent only if during the particular execution the need for the library arises
  - Program can adapt its execution to the fact that some of the libraries is not present in the system (e.g. some functionality is only available to the user if he also has a certain library, etc.)



Linus again sums it up

I think people have this **incorrect picture** that **"shared libraries are inherently good"**. They really really aren't. They cause a lot of problems, and the **advantage really should always be weighed** against those (big) **disadvantages**.

Pretty much the only case **shared libraries really make sense** is **for truly standardized system libraries** that are everywhere, and are part of the base distro.

[Or, for those very rare programs that end up dynamically loading rare modules at run-time - not at startup - because that's their extension model. But that's a **different kind of "shared library"** entirely, even if ELF makes the technical **distinction between "loadable module" and "shared library"** be a somewhat moot point]

Source: https://lore.kernel.org/lkml/CAHk-=whs8QZf3YnifdLv57+FhBi5_WeNTG1B-suOES=RcUSmQg@mail.gmail.com/