Modelo de Compilação e Execução

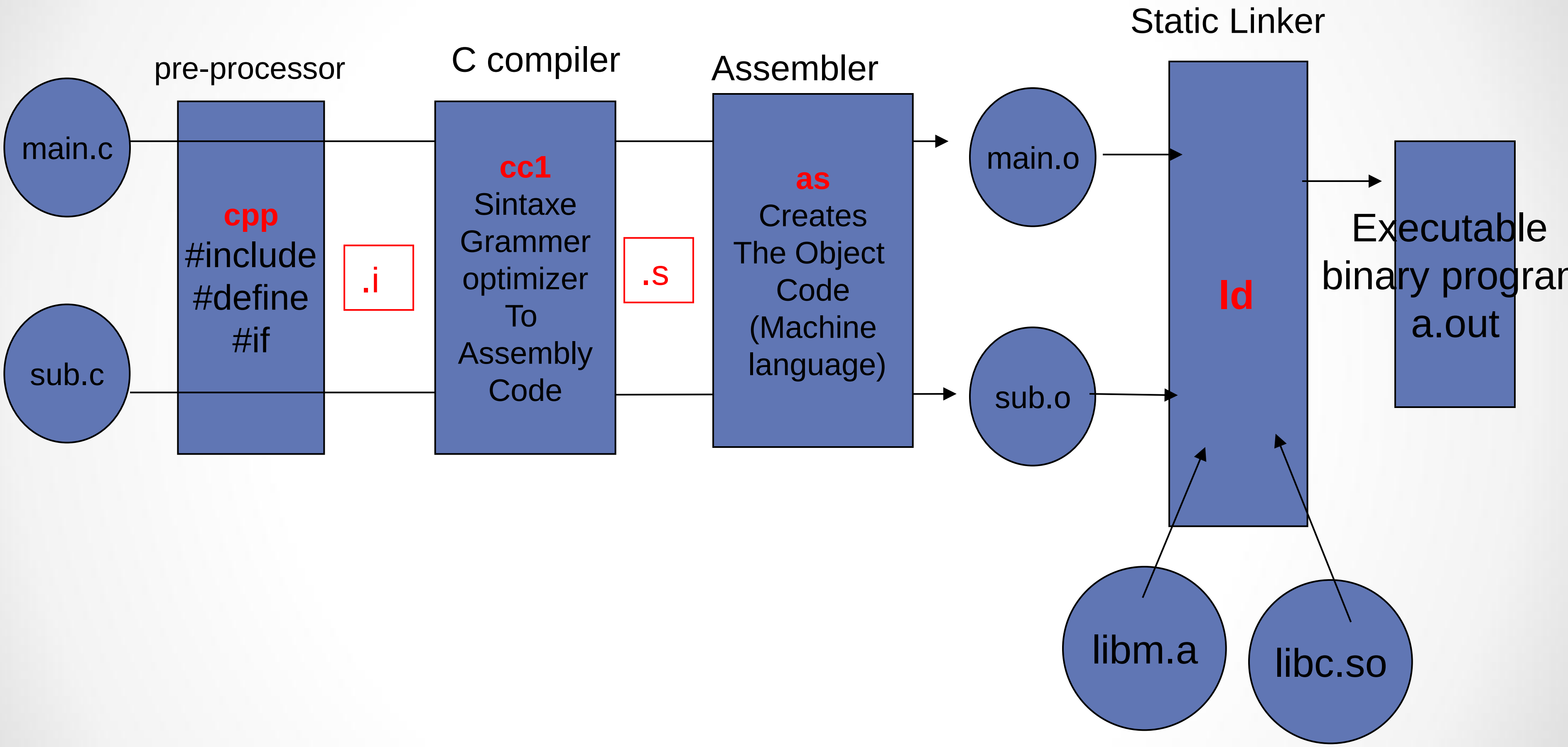
Paul Crocker

Um projecto de 2 Ficheiros

```
/*main.c*/
#include <stdio.h>
float sub();
int  main()
{
   printf(“ola %f\n”,sub());
   return (0);
}
```

```
/*sub.c*/
#include <math.h>
#define   PI  (3.14)
const int quatro=4;
float sub()
{
   #ifdef A
      return ( sqrt (quatro*PI ) );
   #else
      return 1.0;
   #endif
}
```

Compilation Model



main.c

sub.c

pre-processor

cpp
#include
#define
#if

.i

C compiler

cc1
Sintaxe
Grammer
optimizer
To
Assembly
Code

.s

Assembler

as
Creates
The Object
Code
(Machine
language)

main.o

sub.o

Static Linker

ld

Executable
binary progra
a.out

libm.a

libc.so

Static and Dynamic Libraries

Inerir código (static) ou apenas (stub)

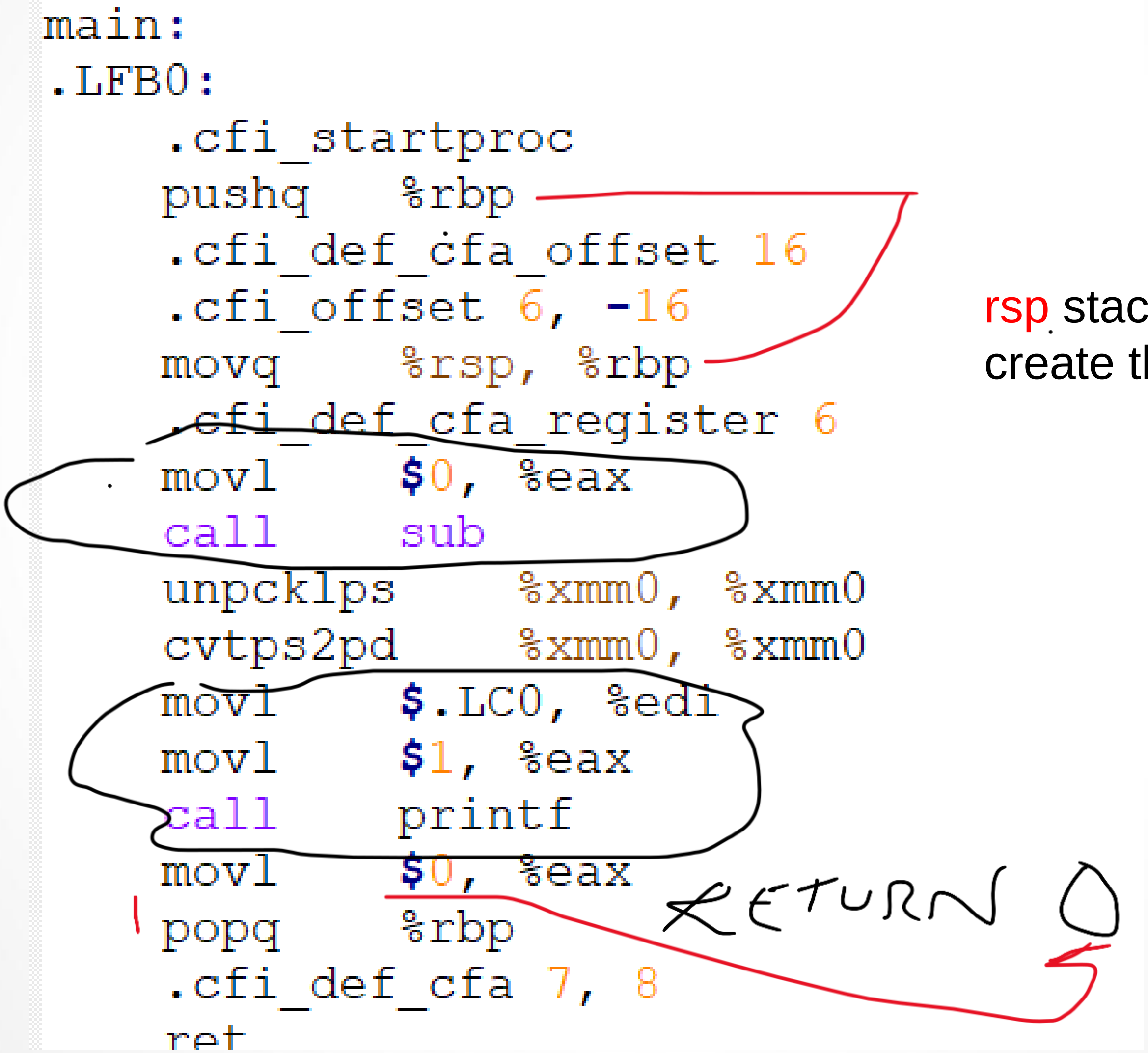
Processo de Compilação

Tarefa	Compilação	Exercícios
Pre-Processor	cc -E main.c -o main.i cc -E sub.c -o sub.i	investigar os ficheiros .i less main.i sub.i O que aconteceu às linhas originais #include ? #define ? const int ?
Pre-Processor Options. Define a constant or Macro	-D PI=3.1	Defina o valor do constante PI do pre processor.
Pre-Processor Options	-D A	Defina o valor do constante A do pre processor. Neste caso sem valor apenas a sua existência.

Processo de Compilação

Tarefa	Compilação	Exercícios
Compiler	cc -Wall -ansi -S *.i	ver os ficheiros (main.s e sub.s) de assembler que são produzidos
Assembler (as)	cc -c *.s	i) Ver tamanhos e tipos dos ficheiros objectos. ls –l *.o e file *.o (ii) Ver symbol table nm main.i e nm sub.o (iii) Ver ficheiros usando o dissasembler objdump Dissambler : objdump- d *.o OU objdump -M intel –d *.o
Linker (-o calls the gnu linker ld)	cc -o exemplo main.o sub.o -lm	(i) Ver symbol table nm example (2) Dissasemble Objdump –d example

Assembler

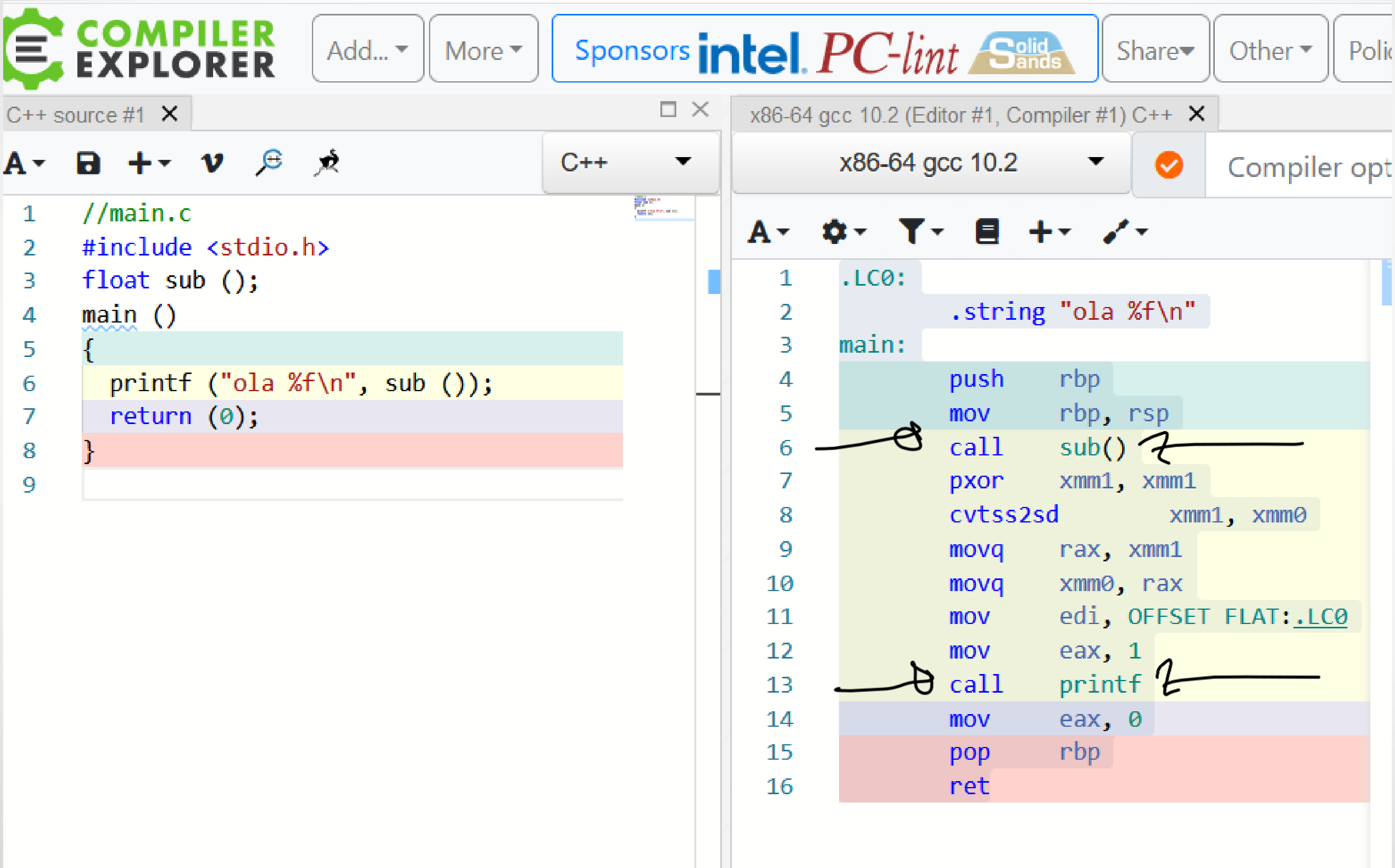


```
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, %eax
        call    sub
        unpcklps        %xmm0, %xmm0
        cvtps2pd        %xmm0, %xmm0
        movl    $.LC0, %edi
        movl    $1, %eax
        call    printf
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
```

RETURN 0

rsp stack pointer
create the stack frame.

Simplify the assembler



```
//main.c
#include <stdio.h>
float sub ();
main ()
{
  printf ("ola %f\n", sub ());
  return (0);
}
```

```
.LC0:
        .string "ola %f\n"
main:
        push    rbp
        mov     rbp, rsp
        call    sub()
        pxor    xmm1, xmm1
        cvtss2sd        xmm1, xmm0
        movq    rax, xmm1
        movq    xmm0, rax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 1
        call    printf
        mov     eax, 0
        pop     rbp
        ret
```

Some Assembler

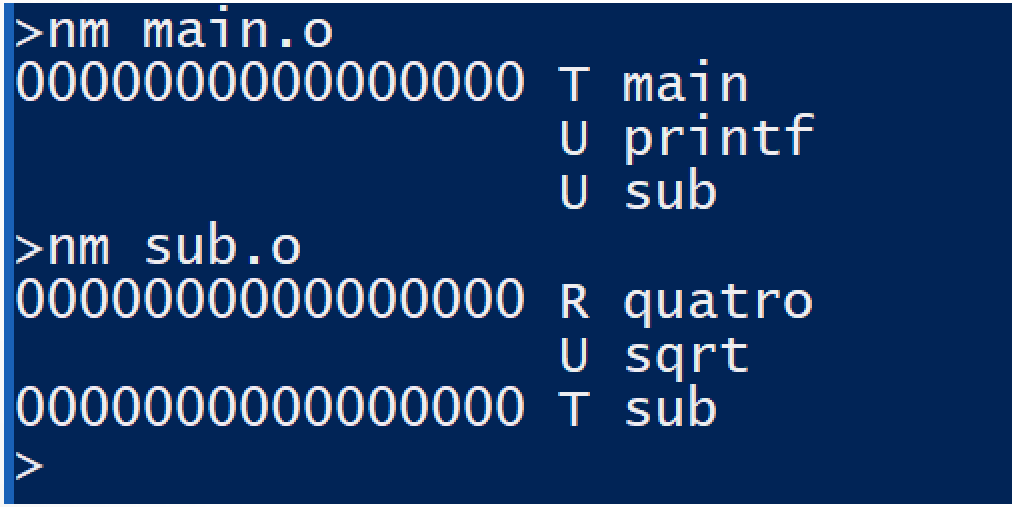
```
main:
   pushq    %rbp          # save %rbp on the stack
   movq     %rsp, %rbp    # store the value of %rsp in %rbp
   movl     $0, %eax      # store the value 0 in %eax
   call     sub           #set PC to sub address
   popq     %rbp          # restore %rbp with the value
                            saved on the stack
   ret              # return from this function
```

Value that look like %rbp or %eax are registers → Memory that is ON the CPU.
%rbp and %rsp are special registers that refer to the base pointer and stack pointer

- PC is the program counter.
- Registers that start with “r” are 64-bits and those with “e” are 32-bits in width.
- The q suffixes on instructions refer to “quad-words” indicating that it is a 64-bit instruction.
- The l suffixes denote 32-bit instructions.

Investigate Object Files

Listing symbols from object file : nm



```
>nm main.o
0000000000000000 T main
                 U printf
                 U sub
>nm sub.o
0000000000000000 R quatro
                 U sqrt
0000000000000000 T sub
>
```

symbols

Disassembling

With obejct files

- objdump –d

```
ubuntu >objdump -d main.o
main.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   b8 00 00 00 00          mov    $0x0,%eax
   9:   e8 00 00 00 00          callq  e <main+0xe>
   e:   0f 14 c0                unpcklps %xmm0,%xmm0
  11:   0f 5a c0                cvtps2pd %xmm0,%xmm0
  14:   bf 00 00 00 00          mov    $0x0,%edi
  19:   b8 01 00 00 00          mov    $0x1,%eax
  1e:   e8 00 00 00 00          callq  23 <main+0x23>
  23:   b8 00 00 00 00          mov    $0x0,%eax
  28:   5d                      pop    %rbp
  29:   c3                      retq
```

Header File Directories

- Especificação de diretorias adicionais para pesquisar ficheiro de inclusão <file.h>
  - –I Additonal Header File Directories (compiler)
  - Linux gcc /usr/inlcude faz sempre parte da lista de diretorias pesquisados para encontrar <file.h>
  - #include "/usr/include/stdio.h   ←→ "#include </usr/include/stdio.h>

Libraries

	Linux	Mac	Windows
Static	.a	.a	.lib
Dynamic	.so	.dylib	.dll

Notation

-lx atalho para libx.so ou libx.a (linker)

-L Additional Library Directories (linker)

Full Linker :   gcc –o example main.o sub.o /usr/lib/libc.so /usr/lib/libm.so

Equivalente :   gcc –o example main.o sub.o -L /usr/lib -lc -lm

… /usr/lib é quase sempre definido por defeito no linker path

Equivalente :   gcc –o example main.o sub.o –lc –lm

…libc.so é incluído sempre por defeito

Equivalente :   gcc –o example main.o sub.o –lm

Nota num Mac –lm não é necessário – incluído por defeito.
/usr/lib pode ser diferente p.ex /usr/lib/x86_64-linux-gnu

Static versus Dynamic

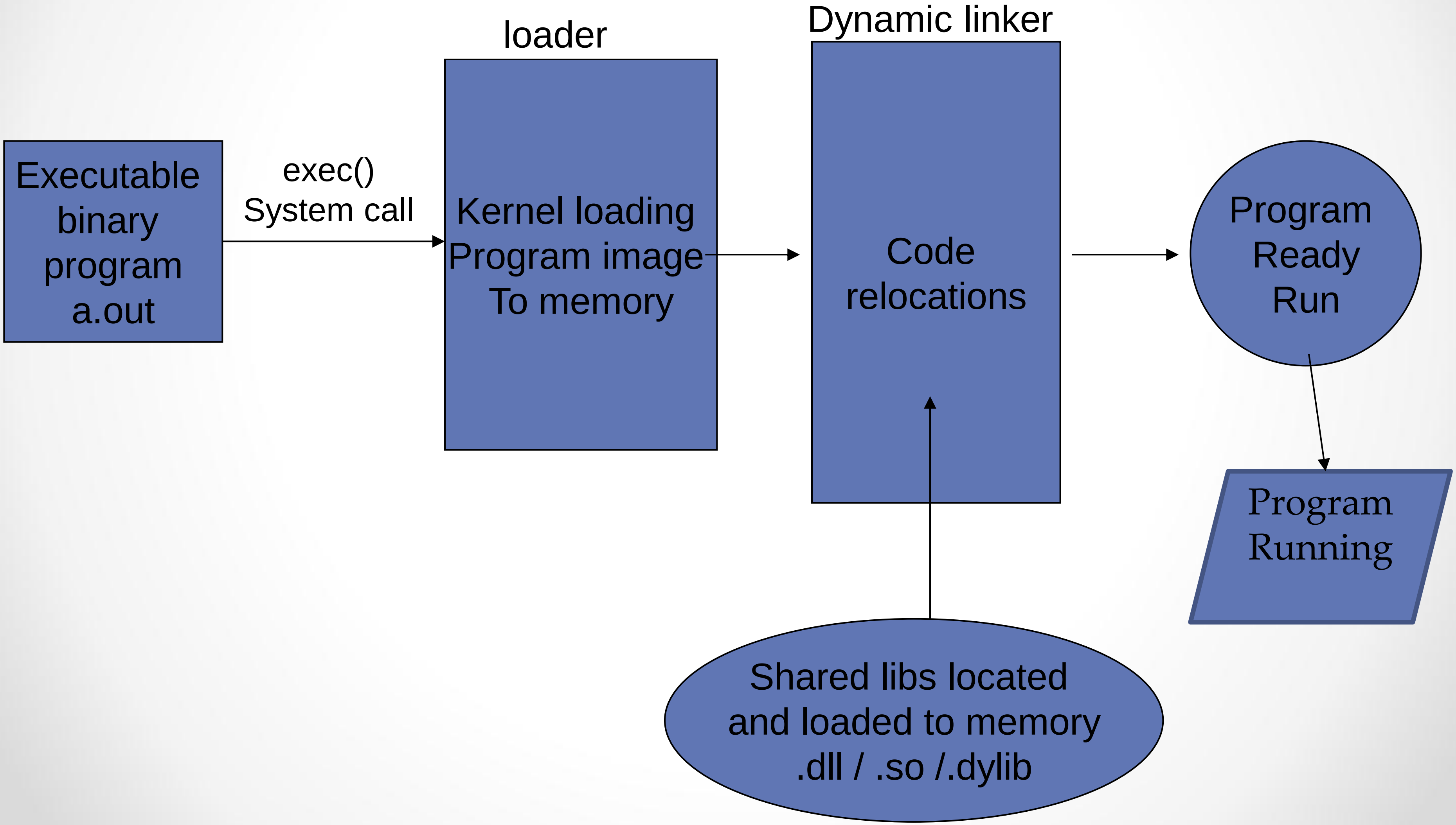
Dynamic Linking

```
>cc -o example main.o sub.o /usr/lib/x86_64-linux-gnu/libm.so
>ls -l example
-rwxrwxrwx 1 crocker crocker 8432 Mar  9 19:29 exemple
```

Static Linking

```
>cc -o example main.o sub.o /usr/lib/x86_64-linux-gnu/libm.a
>ls -l example
-rwxrwxrwx 1 crocker crocker 8552 Mar  9 19:29 exemple
```

Run-Time Model
Dynamic Linking


loader

Dynamic linker

Executable binary program a.out

exec() System call

Kernel loading Program image To memory

Code relocations

Program Ready Run

Program Running

Shared libs located and loaded to memory .dll / .so /.dylib

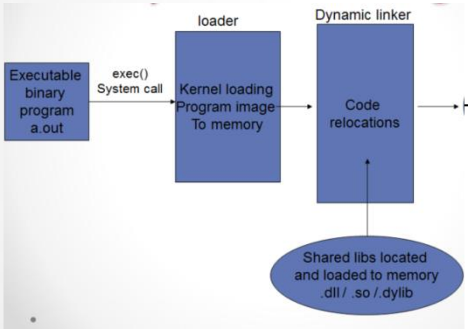
LD.SO(8) Linux Programmer's Manual

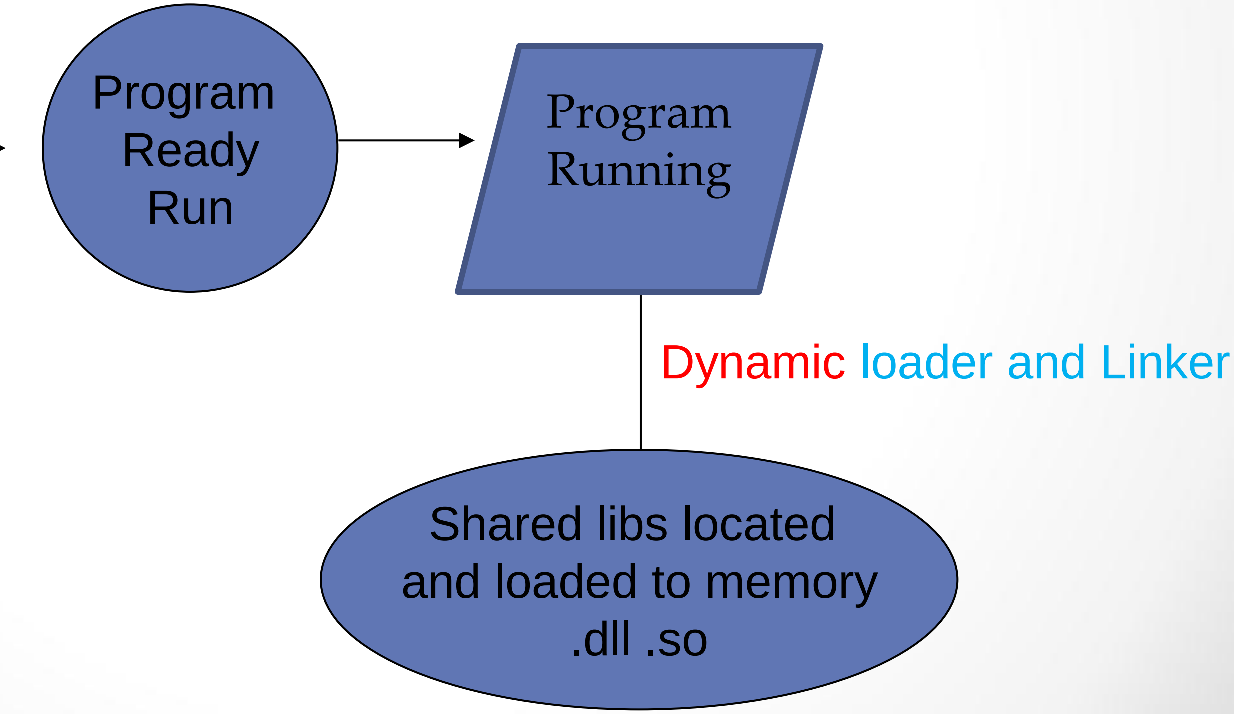
- NAME ld.so, ld-linux.so* - dynamic linker/loader
- DESCRIPTION The programs ld.so and ld-linux.so* find and load the shared libraries needed by a program, prepare the program to run, and then run it.
- Linux binaries require dynamic linking (linking at run time) unless the **-static** option was given to ld during compilation. The program ld.so handles a.out binaries, a format used long ago; ld- linux.so* handles ELF, which everybody has been using for years now.
  - Otherwise both have the same behaviour, and use the same support files and programs ldd(1), ldconfig(8) and /etc/ld.so.conf.

- The shared libraries needed by the program are searched for in various places – in particular
  - By Using the environment variable **LD_LIBRARY_PATH.**
  - etc

Run-Time Model II Dynamic loading

Instead of Linux automatically loading and linking libraries for a given program, it's possible to **share** this control with the application itself i.e Application loads the Libraries during Run Time (NOT Linux at application startup)


loader
Dynamic linker
Executable binary program a.out
exec() System call
Kernel loading Program image To memory
Code relocations
Shared libs located and loaded to memory .dll / .so /.dylib



Program Ready Run

Program Running

Dynamic loader and Linker

Shared libs located and loaded to memory .dll .so

Exemplo Dynamic Loading

```
#include <stdio.h> <dlfcn.h> <string.h>

void invoke_method( char *lib,
                                 char *method, double argument )
{
  void *dl_handle;
  double (*func)(double);
  char *error;

  /* 1 Open the shared object */
  dl_handle = dlopen( lib, RTLD_LAZY );
  if (!dl_handle) { printf( "!!! %s\n", dlerror() );return;}

  /* 2 Resolve the symbol (method) from the object */
  func = dlsym( dl_handle, method );
  error = dlerror();
  if (error != NULL) {printf( "!!! %s\n", error );return;}

 /* 3 Call the resolved method and print the result */
  printf("  %lf\n", (*func)(argument) );

  /* 4 Close the object */
  dlclose( dl_handle );
}
```

```
#define MAX_STRING      80

int main( int argc, char *argv[] )
{
  char line[MAX_STRING+1];
  char lib[MAX_STRING+1];
  char method[MAX_STRING+1];
  double argument;

  while (1) {

    printf("> ");

    line[0]=0;
    fgets( line, MAX_STRING, stdin);

    if ( !strncmp(line, "bye", 3)) break;

    sscanf( line, "%s %s %lf", lib, method, &argument);

    invoke_method( lib, method, argument );
  }
  return 0;
}
```

Utilização

```
cc -o dynload  dynload.c -ldl
cc -o dynload  dynload.c -ldl
./dynload
> libm xx 2
!!! libm: cannot open shared object file: No such file or directory
> libm.so sqrt 4.0              //libm loaded to memory if necessary !
          2.000000
> libm.so cosf 0.0
          1.000000
> libm.so exp 1.0
          2.718282
> bye
```

Reference:
http://www.ibm.com/developerworks/library/l-dynamic-libraries/

Monitoring
Run Time Execution

- Many tools and methods !!

- There are Software and **Hardware** monitors

- Command Line Tools
  - Linux : ps, top
  - Windows Power Shell Equivalentes
    - ps
    - while (1) { ps | sort -desc cpu | select -first 10; sleep -seconds 2; cls }

- Graphical Tools
- Debuggers etc.

Example: strace

- **strace** runs the specified command until it exits.
- It intercepts and records the system calls which are called by a process and the signals which are received by a process.
- The name of each system call, its arguments and its return value are printed on standard error or to the file specified with the -o option
- Each line in the trace contains the system call name, followed by its arguments and its return value.
- An example from stracing the command
  - "cat /dev/null" is:
  - open("/dev/null", O_RDONLY) = 3

Usando strace para ver a utilização dos system calls no printf

- printf writes to a buffer (managed by the c standard library) attached to the file : FILE *stdout.
- The actual writing to disk is done by the **write()** system call

```
printf(“ola 1.0\n”);
Versus
printf(“o”); printf(“l”); printf(“a”); printf(“\n”);
```

- Quantos writes em cada caso ?

Program

```
#include <stdio.h>
int main(){
   printf(“ola\n”);
   printf(“o”); printf(“l”); printf(“a”); printf(“\n”);
   return 0;
}
```

strace

Desligando o buffer do stdout !

- Quantos chamadas a função write() ?

```
#include <stdio.h>
int main(){
   setvbuf(stdout,NULL,_IONBF,0);
   printf(“ola\n”);
   printf(“o”); printf(“l”); printf(“a”); printf(“\n”);
   return 0;
}
```