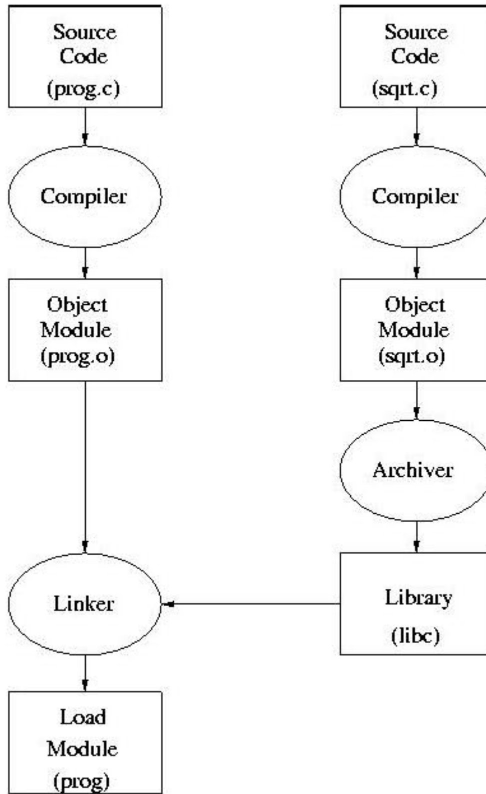
# Compilation and Linking

```
/* p1.c */
int x;
int z;
main()
{
  x=0; z=0;
  printf("f(3)=%d x=%d z=%d\n",
                    f(3),x,z);
}
```

```
gcc -c p1.c
```

- Code for `int f(int)` not available yet (nor `printf`)
- x and z available to other object modules
- Compiled module must reflect these facts

# Compilation and Linking



# Compilation and Linking

Compiler: Converts program from source file to machine language, produces an **object module** (which cannot be executed)

Linker: Produces a **load module** which is ready to be executed

Operating system will create a process from the load module

```
/* p2.c */
static int z;
int f(a)
int a;
{
  extern int x;
  x=14; z=1;
  return(a);
}
```

```
[jmayo@asimov ~/3411]$ gcc -c p1.c
[jmayo@asimov ~/3411]$ gcc -c p2.c
[jmayo@asimov ~/3411]$ file p1.o
p1.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
[jmayo@asimov ~/3411]$ file p2.o
p2.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
[jmayo@asimov ~/3411]$ gcc -o p1 p1.o p2.o
[jmayo@asimov ~/3411]$ file p1
p1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5, dynamically linke
```

- Object file may contain unresolved global *symbols*
  - Defined: variables, functions defined within object file, can be referenced within other object files
  - Undefined: variables, functions used within this object file, defined elsewhere
  - Linker combines object files and resolves symbols while creating executable
    - Object file contains *symbol table*
    - Symbol table will contain information needed to resolve symbols
    - Linker uses information from the symbol table
  - Executable will contain no unresolved symbols

## NAME

nm - list symbols from object files

## SYNOPSIS

```
nm [-a|--debug-syms] [-g|--extern-only]
   [-B] [-C|--demangle[=style]] [-D|--dynamic]
   [-S|--print-size] [-s|--print-armap]
   [-A|-o|--print-file-name]
   [-n|-v|--numeric-sort] [-p|--no-sort]
   [-r|--reverse-sort] [--size-sort] [-u|--undefined-only]
   [-t radix|--radix=radix] [-P|--portability]
   [--target=bfdname] [-fformat|--format=format]
   [--defined-only] [-l|--line-numbers] [--no-demangle]
   [-V|--version] [-X 32_64] [--help]  [objfile...]
```

## DESCRIPTION

GNU nm lists the symbols from object files objfile.... If no object files are listed as arguments, nm assumes the file a.out.

For each symbol, nm shows:

- o The symbol value, in the radix selected by options (see below), or hexadecimal by default.

- The symbol type. At least the following types are used; others are, as well, depending on the object file format. If lowercase, the symbol is local; if uppercase, the symbol is global (external).

  "A" The symbol's value is absolute, and will not be changed by further linking.

  "B" The symbol is in the uninitialized data section (known as BSS).

  "C" The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references.

  ...

  "D" The symbol is in the initialized data section.

  ...

  "T" The symbol is in the text (code) section.

  "U" The symbol is undefined.

```
[jmayo@asimov ~/3411]$ nm -g p2.o
00000000 T f
         U x
[jmayo@asimov ~/3411]$ nm -g p1.o
         U f
00000000 T main
         U printf
00000004 C x
00000004 C z
```

```
/* p1.c */
int x;
int z;
main()
{
  x=0; z=0;
  printf("f(3)=%d x=%d z=%d\n",
                    f(3),x,z);
}
/* p2.c */
static int z;
int f(a)
int a;
{
  extern int x;
  x=14; z=1;
  return(a);
}
```

# Object Modules

e.g. p1.o, p2.o

Many different formats (a.out, ELF, COFF, etc.)

**Header Section** - Sizes required to parse object module and create program

**Machine Code** - Generated machine code (also called *text*)

**Initialized Data** - Initialized global and static data (doesn't go on stack)

**Symbol Table** - External symbols

*Undefined* - Used in this module, defined elsewhere
*Defined* - Defined in this module, may be undefined in another module

**Relocation Information** - Record of places where symbols must be relocated

```
#include <iostream.h>
#include <math.h>

float arr[100];
int   size=100;

void main( int argc, char *argv[] ) {

    int i;
    float sum = 0;

    for ( i = 0; i < size; ++i ) {

        cin >> arr[i];
        arr[i] = sqrt( arr[i] );
        sum += arr[i];

    }

    cout << sum;

}
```

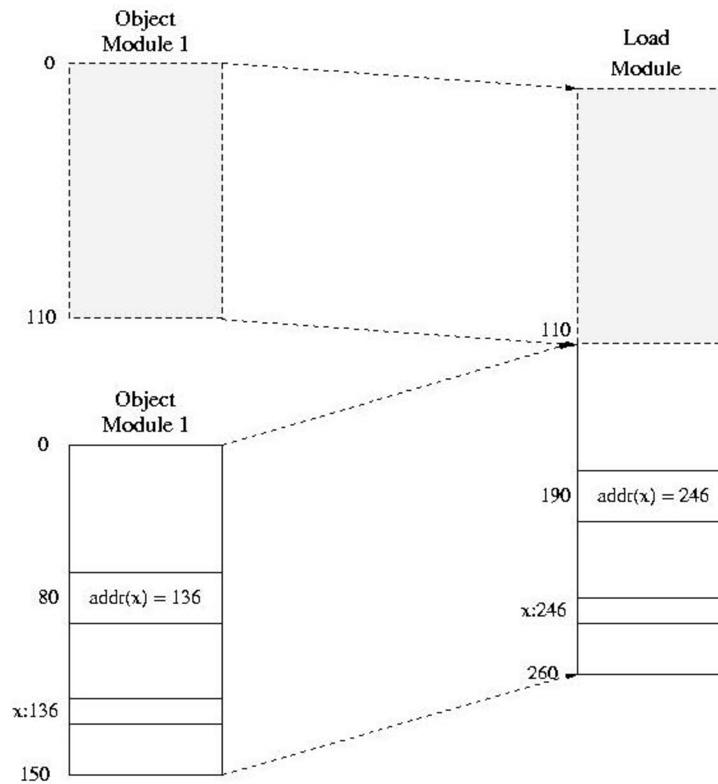
| Offset | Contents | Comment |
|---|---|---|
| | | ***Header Section*** |
| 0 | 94 | Number of bytes of machine code |
| 4 | 4 | Number of bytes of initialized data |
| 8 | 400 | Number of bytes of uninitialized data |
| 12 | 72 | Number of bytes of symbol table |
| 16 | ? | Number of bytes of relocation information |
| | | ***Machine Code Section (text)*** |
| 20 | XXXX | Code for top of `for` loop |
| 50 | XXXX | Code for `arr[i] << cin` |
| 66 | XXXX | Code for `arr[i]=sqrt(arr[i])` |
| 86 | XXXX | Code for `sum += arr[i]` |
| 98 | XXXX | Code for bottom of `for` loop |
| 102 | XXXX | Code for `cout << sum` |
| | | ***Initialized Data Section*** |
| 114 | 100 | Location of `size` |
| | | ***Symbol Table Section*** |
| 118 | ? | `"size"` = 0 (in data section) |
| 130 | ? | `"arr"` = 4 (in data section) |
| 142 | ? | `"main"` = 0 (in code section) |
| 154 | ? | `">>"` = external , used at 42 |
| 166 | ? | `"sqrt"` = external, used at 62 |
| 178 | ? | `"<<"` = external, used at 90 |
| | | ***Relocation Information Section*** |
| 190 | ? | Relocation Information |

Figure taken from *Operating Systems: A Design-Oriented Approach*, Charles Crowley, Irwin, 1997

# Linking

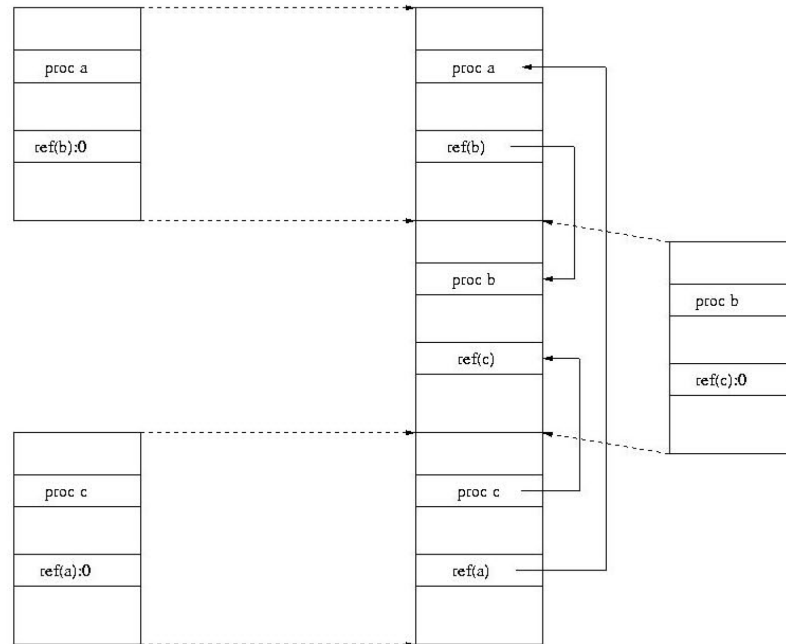
- Object module will (usually) assume starting address is zero
- Linker combines several object modules
  - ⋆ Text sections combined, data sections combined, ...
- Combined modules cannot all start at zero
- Cannot have unresolved references in load module
- Two tasks then:
  - ⋆ Relocate modules (account for starting address that results from combining modules)
  - ⋆ Link modules (resolve undefined external references)

# Relocation[1]




[1]Figure taken from *Operating Systems: A Design-Oriented Approach*, Charles Crowley, Irwin, 1997

# Linking[2]




[2] Figure taken from *Operating Systems: A Design-Oriented Approach*, Charles Crowley, Irwin, 1997

# Load Module Creation

(1) Create empty load module and global symbol table

(2) Get next object module or library name

(3) Object module:

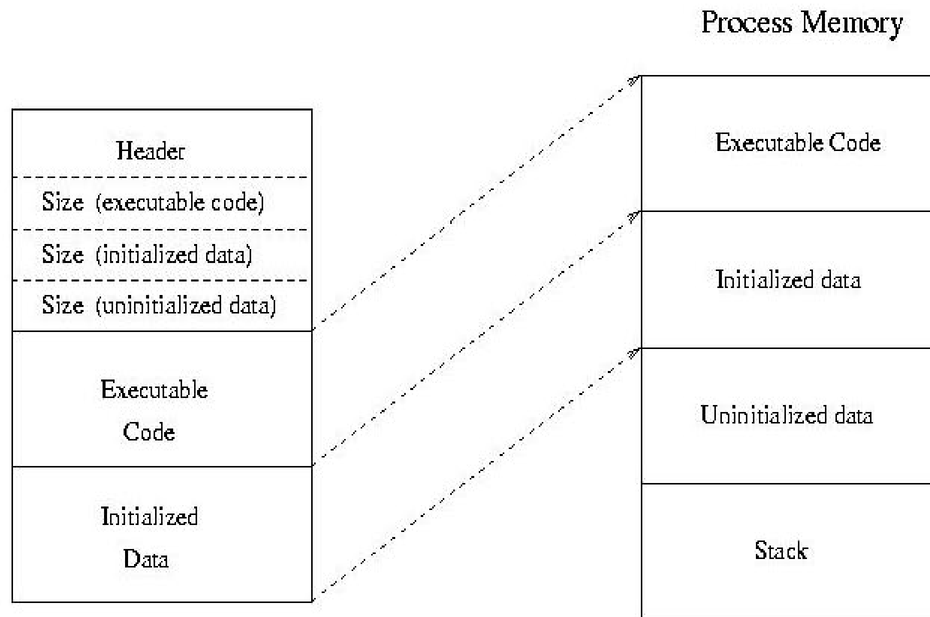
   (a) Insert code and data, remember where
   (b) Relocate object module and all symbols in module's symbol table
   (c) *Undefined external references:*
      - Already defined in global symbol table, write value in just loaded object module
      - Not yet defined, note that links must be fixed when symbol defined
   (d) *Defined external references:*
      - Fix up all previous references (to this symbol) noted in global symbol table

(4) Library:
   - ⋆ Find each undefined external reference in global symbol table
   - ⋆ See if symbol defined in library
   - ⋆ If so, load it per step (3)

(5) Back to step 2

- Load module need not contain relocation (in most cases) or symbol table sections
- Symbol table information may be used by debugger or *stripped* to reduce binary size

# Process Creation[3]




[3] Figure taken from *Operating Systems: A Design-Oriented Approach*, Charles Crowley, Irwin, 1997

## Static Linking

- Library routines combined into binary program image
- Creates large load modules
- Same library may be contained in multiple images throughout file system
- Once load module is created, it is impervious to changes in referenced library
  - ⋆ Cannot incorporate new versions without recompilation
  - ⋆ Does not depend on existence of (specific version of) library on system
- `gcc -static ...`

# Dynamic Linking

- *Stub* included in binary program image for each library-routine reference
- Stub is code to locate memory-resident routine or load it if library routine not present
- Stub replaces itself with address of routine and executes routine
- Will use most recent version of library routine
- Higher overhead during use; faster startup than statically linked
- Allows same code to be shared by multiple process
- All processes using a language library execute single copy of library code (shared library)
- Generally requires help from OS (code mapped into multiple address spaces)
- More efficient use of memory

# Static Libraries

- Static libraries created with `ar`. See manual page. Commonly used options:

| | |
|---|---|
| c | create a new library |
| q | add the named file to the end of the archive |
| r | replace a named archive/library member |
| t | print a table of archive contents |

- `ranlib` run on library to create index of each symbol defined by a relocatable object in library
- Example:

```
gcc -c libFunc.c
ar -cq libMyLib.a libFunc.o
ranlib libMyLib.a
gcc myProg -lmyLib
gcc -o myProg myProg.c -L. -lMyLib
ar -q libMyLib.a anotherFunc.o
ranlib libMyLib.a
```

- See also `objdump`

# Determining Library Contents

```
[jmayo@asimov hw2]$ cat functionA.c
extern int x;
int functionA(int a,int b)
  int c;
  c=a+b+x;
  return(c);

[jmayo@asimov hw2]$ cat functionB.c
int v;
static int w;
int functionB(int x,int y)
  int z;
  z= -w-x-y;
  return(z);
```

```
[jmayo@asimov hw2]$ gcc -c functionA.c
[jmayo@asimov hw2]$ gcc -c functionB.c
[jmayo@asimov hw2]$ ar -cq libEx.a functionA.o
[jmayo@asimov hw2]$ ar -q libEx.a functionB.o
[jmayo@asimov hw2]$ nm -g libEx.a

functionA.o:
00000000 T functionA
         U x

functionB.o:
00000000 T functionB
00000004 C v
```

```
[jmayo@asimov hw2]$ nm -s libEx.a

Archive index:
functionA in functionA.o
functionB in functionB.o
v in functionB.o

functionA.o:
00000000 T functionA
         U x

functionB.o:
00000000 T functionB
00000004 C v
00000000 b w
[jmayo@asimov hw2]$
```

# Dynamic Libraries

- Must compile *position independent code*; `gcc -fPIC -c myFunc.c`
- Use `ld` to create library; `gcc -shared *.o -o libmyUtil.so` (ld via gcc)
- `ldd` returns shared libraries used by an object module

- Example:

```
gcc -fPIC -c p2.c
gcc -shared p2.o -o libmyUtil.so
gcc -o p1 p1.c -L. -lmyUtil

./p1
f(3)=3 x=14, z=0

ldd simpleFunc
    libmyUtil.so => /home/csdept/jmayo/3411/libmyUtil.so
    libc.so.6 => /lib/i686/libc.so.6
    libgcc_s.so.1 => /lib/libgcc_s.so.1
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2
```

```
[jmayo@asimov shared]$ gcc -fPIC -c functionA.c
[jmayo@asimov shared]$ gcc -fPIC -c functionB.c
[jmayo@asimov shared]$ gcc -shared *.o -o libEx.so
[jmayo@asimov shared]$ nm -g libEx.so
00001908 A _DYNAMIC
000019e4 A _GLOBAL_OFFSET_TABLE_
         w _Jv_RegisterClasses
00001a1c A __bss_start
         w __cxa_finalize@@GLIBC_2.1.3
         w __deregister_frame_info_bases@@GCC_3.0
000018fc D __dso_handle
         w __gmon_start__
         w __register_frame_info_bases@@GCC_3.0
00001a1c A _edata
00001a40 A _end
000008e0 T _fini
00000678 T _init
00000820 T functionA
00000858 T functionB
00001a3c B v
         U x
```