How Are Compiled Programs Stored On Disk?

Let us examine the creation and initial execution of a new process, first by examining how its program is stored on disk, and then how it is loaded and represented in memory.

Consider a simple program, compiled and linked from two object files. This program is written in C, but we could examine any program that is executed directly, *natively*, by the processor.

Here is the file *program.c*

```c
#include <stdio.h>

#define MAX_VALUE       100

static  int     one_max = MAX_VALUE;
static  int     other_max[MAX_VALUE];

static void my_square(int x)
{
    extern void print_sqrt(int);
    int square_local;

    other_max[0]        = one_max;
    x   = x*x;

    print_sqrt(x);
}

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

    for(main_i=0 ; main_i<MAX_VALUE ; ++main_i)
        my_square(main_i);

    return(0);
}
```

Here we'll examine how a program the compiled, linked and loaded in the LINUX environment. Repeat these demonstrations with your own multi-file programs.

Examining The Resultant Object File

We can next compile this single source file into an *object file* or *object module*. This module is not yet executable, as it lacks (at least) the external function `print_sqrt()`. Still, we can examine this *Extensible Linking Format* (ELF) file in a number of ways:

```
prompt> gcc -c program.c

prompt> objdump -x program.o

program.o:     file format elf32-i386
program.o
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000066  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  00000000  00000000  0000009c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000190  00000000  00000000  000000a0  2**2
                  ALLOC
  3 .note         00000014  00000000  00000000  000000a0  2**0
                  CONTENTS, READONLY
  4 .comment      00000014  00000000  00000000  000000b4  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
00000000 l    df *ABS*   00000000 program.c
00000000 l    d  .text   00000000
00000000 l    d  .data   00000000
00000000 l    d  .bss    00000000
00000000 l       .text   00000000 gcc2_compiled.
00000000 l     O .data   00000004 one_max
00000000 l     F .text   00000028 my_square
00000000 l     O .bss    00000190 other_max
00000000 l    d  .note   00000000
00000000 l    d  .comment     00000000
00000000         *UND*   00000000 print_sqrt
00000028 g     F .text   0000003e main
```

You may also like to try the *nm*, *readelf*, and *size* programs.

Examining The Resultant Object File, *continued*

The object file consists of a number of *sections* for containing the object's instructions and data (the `.text` and `.data` sections), and describing the object (the `.bss` and `.symtab` sections).

If brave enough, we can also attempt to *disassemble* the instructions from the `.text` segment of the file. Clearly this activity is highly dependent on the hardware architecture being used, here for an Intel x86 processor.

```
prompt> objdump -d program.o

program.o:     file format elf32-i386

Disassembly of section .text:

00000000 <my_square>:
   0:     55                  pushl  %ebp
   1:     89 e5               movl   %esp,%ebp
   3:     83 ec 04            subl   $0x4,%esp
   6:     a1 00 00 00 00      movl   0x0,%eax
   b:     a3 00 00 00 00      movl   %eax,0x0
  10:     8b 55 08            movl   0x8(%ebp),%edx
  [lots omitted from here ....]
  1e:     e8 fc ff ff ff      call   1f <my_square+1f>
  23:     83 c4 04            addl   $0x4,%esp
  26:     c9                  leave
  27:     c3                  ret

00000028 <main>:
  28:     55                  pushl  %ebp
  29:     89 e5               movl   %esp,%ebp
  [lots omitted from here ....]
  64:     c9                  leave
  65:     c3                  ret
```

Other LINUX tools list the object's symbol table (see *man nm*), and dump the full file in varying formats (see *man od*).

The Second Module For Our Program

Here is the file *printer.c*

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

void print_sqrt(int x)
{
    printf("sqrt(%d) = %f\n", x, sqrt((double)x) );
}
```

We can again examine the symbol table of this new object module, and look at the size of both objects:

```
prompt> gcc -c printer.c

prompt> objdump -t printer.o

printer.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    df *ABS*   00000000 printer.c
00000000 l    d  .text   00000000
00000000 l    d  .data   00000000
00000000 l    d  .bss    00000000
00000000 l       .text   00000000 gcc2_compiled.
00000000 l    d  .rodata 00000000
00000000 l    d  .note   00000000
00000000 l    d  .comment     00000000
00000000 g     F .text   0000002d print_sqrt
00000000         *UND*   00000000 sqrt
00000000         *UND*   00000000 printf

prompt> size *.o
text    data    bss     dec     hex     filename
60      0       0       60      3c      printer.o
102     4       400     506     1fa     program.o
```

The Role Of The Linker

The role of the *linker*, or *linkage editor*, is to now *resolve* any undefined references between the object modules, and build a single executable program.

Clearly, `program.o` requires the function `print_sqrt()`, but `printer.o` also demands `printf()`, `exit()` and `sqrt()`.

The `printf()` is one of the standard C functions, and because these are so common, we need not indicate to the linker that we wish to link against their standard C library (on our LINUX machines this is the 1.2MB `/lib/libc.so.6`).

However, `sqrt()` is strictly a C function found in C's maths library (where is this?). We thus need to indicate the need to also link with the maths library:

```
prompt> gcc -o program program.o printer.o -lm
```

Deferred Linking

Perhaps surprisingly, the linker does not *fully* resolve all references at this point. All references to symbols found within our our object files are bound to form a single executable program.

However, symbols such as `printf()`, `exit()` and `sqrt()` are only checked for their validity, to ensure that they do exist somewhere, and are not (yet) combined into our program. We say that their linking is *deferred* until required. The linker stores in our executable program, the location of external libraries where it found the "missing" symbols:

```
prompt> ldd program
        libm.so.6 => /lib/i686/libm.so.6 (0x4001d000)
        libc.so.6 => /lib/i686/libc.so.6 (0x42000000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```


The Complete Executable Program

```
prompt> ls -l program
-rwx------   1 chris    chris        4609 Aug 16 23:14 program

prompt> size program
text    data    bss     dec     hex     filename
1254    216     404     1874    752     program

prompt> nm -t d -n program
         U __gmon_start__
         U __libc_init_first
         U atexit
         U exit
         U printf
         U sqrt
134513600 ? _init
134513756 T _start
134513824 t __do_global_dtors_aux
134513864 t fini_dummy
134513872 t my_square
134513912 T main
134513976 T print_sqrt
134514024 t __do_global_ctors_aux
134514064 t init_dummy
134514072 A _etext
134514080 ? _fini
134518220 D __data_start
134518220 W data_start
134518224 d force_to_data
134518224 d one_max
134518228 ? __CTOR_LIST__
134518228 d force_to_data
134518232 ? __CTOR_END__
134518236 ? __DTOR_LIST__
134518240 ? __DTOR_END__
134518244 A _GLOBAL_OFFSET_TABLE_
134518284 A _DYNAMIC
134518436 A __bss_start
134518436 B __environ
134518436 A _edata
134518436 W _environ
134518440 b other_max
134518840 A _end
```

Notice that there are many additional symbols that we did not define in our program.

Loading A Program From Disk

We now have a successfully compiled and linked program on disk. How shall the operating system commence execution of this file's contents?

We remember that the LINUX system call `execve()` (and its simplifying interfaces) are used to overlay a *new* program into the *current* process. This operation requires two important tasks:

- Memory and access validation –
  the operating system kernel first reads in the program file's *header* information and validates it for type, access permissions, memory requirements, and its ability to run its instructions.

  It confirms that file is an executable image, and calculates memory requirements. Any errors at this point indicate that "Execute permission is denied for the file".

```
prompt> file ./pxref
pxref:     sparc demand paged dynamically linked executable
prompt> ./pxref
zsh: exec format error: ./pxref
```

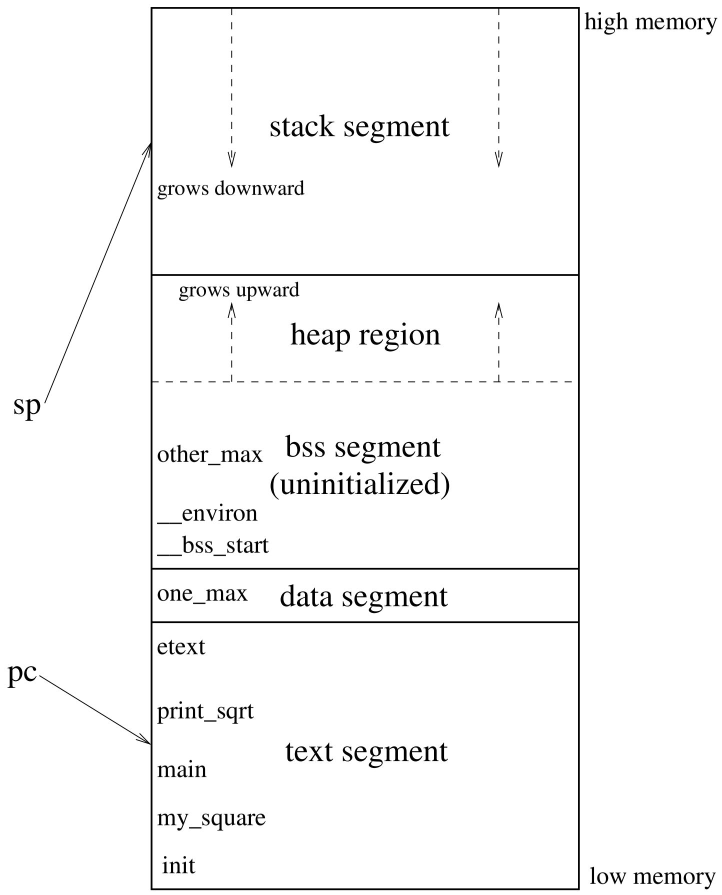
- Process setup – the kernel now uses the header's information to direct the start address for process execution (typically the first executable address in the file), sizes and virtual addresses for each section of text and data, and sizes and virtual addresses for symbolic debugging information (if required).

  New memory is next allocated to hold the text (read-only). New memory areas are allocated for data initialized at compile-time (contents copied from file), and then grown to hold uninitialized data (the *.bss*) and the initial stack.

  Each section of memory required will be placed on a *page boundary*, i.e. a multiple of 4KB.

How The Process Is Stored In Memory

We can now see how the operating system kernel will load the *process image* into physical memory.



Remember that all memory references will be made relative to the beginning of the process's image, permitting the process to be loaded anywhere in physical memory.

Dynamic Linking Of External Symbols

What happened to the symbols `printf()`, `exit()` and `sqrt()`? Our linking phase chose to *defer* their resolution, leaving unresolved references in our program.

The operating system may resolve these at one of two times:

- **Load-time dynamic linking** – the application program is read from disk into memory and unresolved references are located.

  The load-time loader finds *all* necessary (external) symbols, in our case the `sqrt()` function from the maths library, and alters *all* references to each symbol (all previously zero) to memory references relative to the beginning of the program.

  All unresolved symbols are sought and loaded, regardless of whether they are required/called at run-time.

- **Run-time dynamic linking** (or *lazy linking*) – the application program is read from disk into memory and unresolved references are left as invalid (typically zero).

  The first access of an invalid, unresolved, reference results in a *software trap*. The run-time dynamic linker determines why this trap occurred, and seeks the necessary (external) symbol.

  Only this symbol is loaded into memory and linked into the calling program.

How does the linker know where to look? Examine the output of

```
prompt> strings program
```

and you can see the names of C libraries that LINUX's run-time loader will search for.

The Advantages Of Dynamic over Static Linking

It may appear inefficient to defer the linking of all symbols until they are finally needed (why not link completely at compile/link time?).

There are significant advantages to deferring linking over traditional *static linking*:

- Program files (on disk) become *much* smaller because they need not hold all necessary text and data:

```
prompt> gcc -o program program.o printer.o -lm
prompt> ls -l program
-rwx------   1 chris    chris        4609 Aug 17 00:51 program
prompt> gcc -static -o program program.o printer.o -lm
prompt> ls -l program
-rwx------   1 chris    chris      458954 Aug 17 00:51 program
```

- Standard libraries may be upgraded/patched without every old program needing to be re-linked. This clearly requires some agreed module naming convention that enables the dynamic linker to find the newest, installed module (i.e. some *version* specification).

- Software vendors need only provide the libraries (modules) required (or purchased) by the customer, e.g. `GENERAL-LEDGER`, `ACCOUNTS-RECEIVABLE`, but not `TRIAL-BALANCE`. Additional run-time linking functions allow such programs to *programatically-link* required modules (see *man dlopen*).

- In combination with virtual memory, dynamic linking permits two or more processes to *share* read-only modules. Using this technique, only one copy of a module needs be resident in memory at any one time, and multiple processes each execute its (read-only) code. This results in a *considerable* saving of memory, but demands a smarter swapping policy.

Processes Sharing Memory Segments

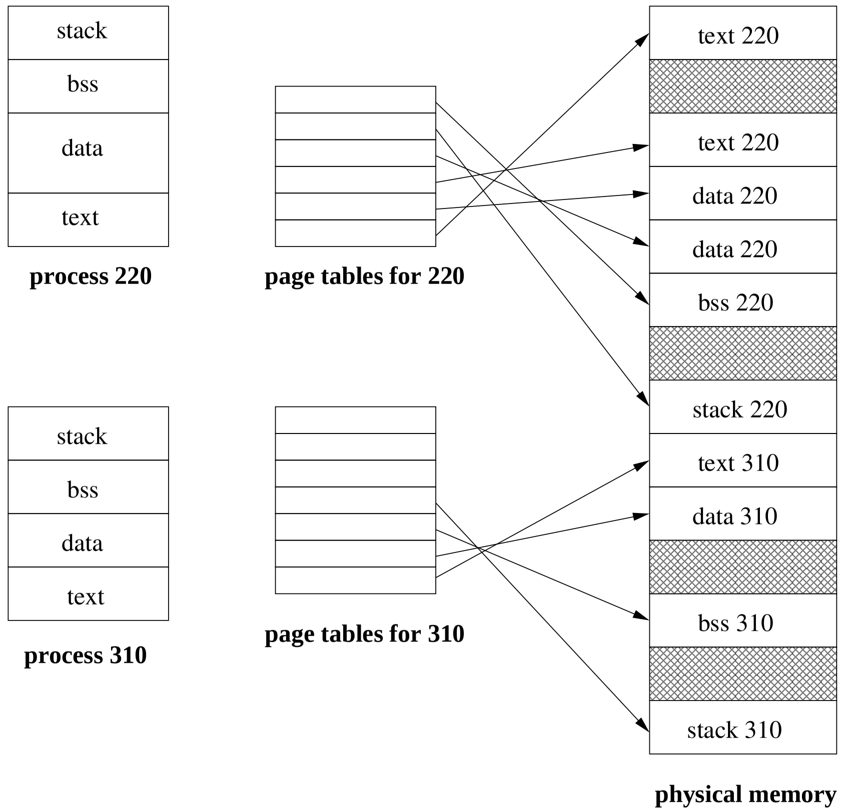
If many programs can now share executable modules (such as standard C and maths libraries), there needs be a strong relationship between the run-time linker and memory management.

```
prompt> /cslinux/examples/cs205/memstat
     68k: PID     1 (/sbin/telinit)
     64k: PID    29 (/sbin/kerneld)
     72k: PID   132 (/usr/sbin/syslogd)
    216k: PID   141 (/usr/sbin/klogd)
     76k: PID   152 (/usr/sbin/crond)
     68k: PID   163 (/usr/sbin/portmap)
     60k: PID   174 (/usr/sbin/inetd)
    256k: PID   188 (/usr/sbin/sendmail)
    216k: PID   200 (/usr/sbin/httpd)
     56k: PID   221 (/sbin/mingetty)
    144k: PID   222 (/usr/X11R6/bin/xdm)
     52k: PID   224 (/sbin/update)
   2764k: PID   226 (/usr/X11R6/bin/XF86_SVGA)
    340k: PID   227 (/usr/X11R6/bin/xdm)
    328k: PID   239 (/usr/X11R6/bin/fvwm95-2)
    696k: PID   261 (/usr/X11R6/bin/nxterm)
    180k: PID   266 (/usr/X11R6/bin/xload)
    240k: PID   267 (/bin/zsh)
    212k: PID   302 (/usr/bin/vi)
    100k: PID  1692 (./memstat)
    288k: /usr/bin/vi 302
    328k: /bin/zsh 267
      8k: /usr/lib/libtermcap.so 261 267 302
     40k: /lib/ld-2.0.6.so 1 29 132 141 152 163 174 188 200 221 ...
     20k: /lib/ld-linux.so.1.9.5 239 262 263 265
    596k: /lib/libc-2.0.6.so 1 29 132 141 152 163 174 188 200 ...
     52k: /usr/lib/libdb.so 188 200
      8k: /usr/lib/libdl.so 222 226 227
     96k: /usr/lib/libm.so 226
     44k: /usr/lib/libresolv.so 188
      8k: /usr/lib/libutil.so 1
 [lots more omitted ...]
--------
  73728k
```

Some sections of memory are now occupied by shared modules, and not just by processes. These sections may only be swapped out with care if other resident processes depend on them.

Memory Segments and Page Tables

Now, let us examine 2 processes, pid=`220` and pid=`310`, occupying pages in physical memory (not related to details on the previous page):

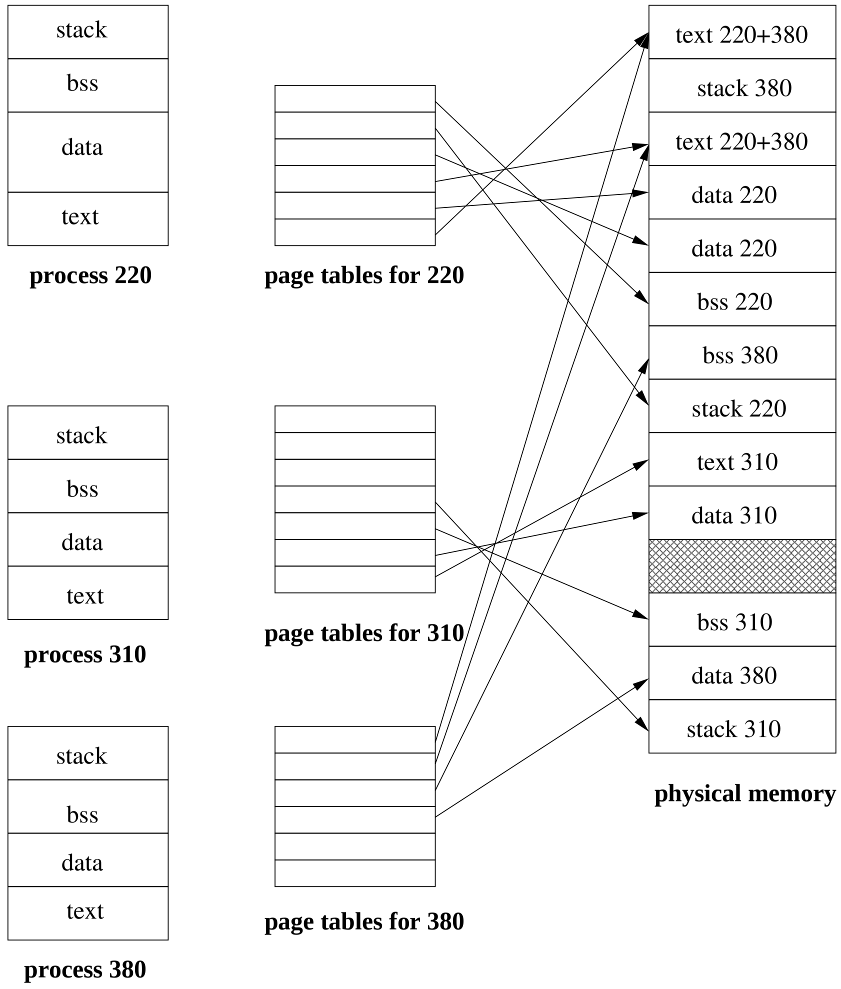


Depending on the required sizes of each process's segments, they may each occupy many pages of physical memory.

Notice that with paging and swapping, the process' memory is not necessarily contiguous. The operating system's memory management simply allocates whatever pages are unallocated at the time.

Process Forking and Memory Usage

Now, imagine that pid=`220` invokes a `fork()` system call to create a new process, pid=`380`. Any read-only pages, such as the process' text segments may now be shared, but data segments must be allocated and copied.



Clearly the operating system must now maintain a reference count of how many processes are using each page.