Link 6.A Loading

Young W. Lim

2019-05-01 Wed

Outline

Based on

"Self-service Linux: Mastering the Art of Problem Determination",
Mark Wilding
"Computer Architecture: A Programmer's Perspective",
Bryant & O'Hallaron

Compling 32-bit program on 64-bit gcc

- `gcc -v`
- `gcc -m32 t.c`
- `sudo apt-get install gcc-multilib`
- `sudo apt-get install g++-multilib`
- `gcc-multilib`
- `g++-multilib`
- `gcc -m32`
- `objdump -m i386`

the memory image

1. invoking the loader
2. loading
3. run-time memory image
4. creating the memory image

invoking the loader

- the shell *runs* an executable object file by *invoking* some memory resident os code known as the `loader`
- any program can *invoke* the `loader` by calling the `execve` function

loading

- the process of copying the program into memory and then running it, is known as `loading`
- the `loader` copies the *code* and the *data* in the executable object file from disk into memory
- then runs the program by jumping to its first instruction (`entry point`)

run-time memory image (1)

- the code segment always starts at address `0x08048000`
- the data segment follows at the next `4-KB` aligned address
- the runtime heap follows on the first `4-KB` aligned address past the read/write segment grows up via calls to the `malloc` library
- shared libraries starts at address `0x40000000`

run-time memory image (2)

- the user stack always starts
  at address 0xbfffffff
  and grows down (towards lower memory addresses)
- the segment starting above the stack
  at address 0xc0000000
  is reserved for the code and data
  in the memory resident part of
  the operating system (kernel)

Linux Run-time Memory Image (1)

- Kernel .... `0xc0000000`
- User Stack .... `%esp`
- Shared Libraries .... `0x40000000`
- Run-time Heap .... `brk`
- Read/Write segment
- Read-only segment .... `0x08048000`
- Unused .... `0x00000000`

Linux Run-time Memory Image (2)

- Kernel . . . . `0xc0000000`
  - memory invisible to user code
- User Stack . . . . `%esp`
  - created in run time
  - grows toward decreasing addresses
- Shared Libraries . . . . `0x40000000`
  - grow toward increasing addresses
- Run-time Heap . . . . `brk`
  - created by `malloc`

Linux Run-time Memory Image (3)

- Read/Write segment
  - `.data` and `.bss`

  -loaded from the executable file
- Read-only segment 0x08048000
  - `.init`, `.text`, `.rodata`

  -loaded from the executable file
- Unused 0x00000000

Linux Run-time Memory Image (4)

Kernel Virtual Memory	Memory invisible to user code	0xc0000000
User Stack	created at run time	%esp
Shared Libraries		0x40000000
Run-time Heap	created by malloc	brk
Read/Write segment	.data, .bss	
Read-only segment	.init, .text, .rodata	0x08048000
Unused		0x00000000

Linux Run-time Memory Image (5)

0xc000_0000	Kernel virtual memory	memory invisible to the user code
	User stack created at run time	← %esp stack ptr
	↓ ↓ ↓ ↑ ↑ ↑	
0x4000_0000	memory mapped region for shared libraries	
	↑ ↑ ↑	
	Run time heap created by malloc	← brk
	R/W segment (.data, .bss)	
0x0804_8000	RO segment (.init, .text, .rodata)	

default address

- most of the time, the various sections do not need to be placed in a specific location
- what matters more is the layout.
- noways, the stack top is actually randomised
- Note that the start of the heap is also randomised.

- `0x08048000` is the default address on which `ld` starts the first `PT_LOAD` segment on `Linux/x86`
- On `Linux/amd64` the default is `0x400000`

https://stackoverflow.com/questions/14795164/why-do-linux-program-text-sections

changing default address

- you can change the default by using a custom linker script
- also can change where .text section starts
  with the `Wl,-Ttext,0xNNNNNNNN flag`

https://stackoverflow.com/questions/14795164/why-do-linux-program-text-sections

the start address is not zero

- `.text` is not mapped at address 0
- the NULL pointer is usually mapped to ((void *) 0) for convenience
- It is useful that the zero page is mapped inaccessible
  to trap uses of NULL pointers.
- The memory before the start of .text is actually
  used by a lot of things;
- cat /proc/self/maps as an example:
  C library, the dynamic loader ld.so and
  the kernel VDSO (kernel mapped dynamic code library that provides
  some interfaces to the kernel).

https://stackoverflow.com/questions/14795164/why-do-linux-program-text-sections

the startup code

1. creating the memory image
2. jumping to the entry point
3. the `crt1.o` startup routine
4. Startup code
5. forking child process
6. invoking the loader
7. deferring copying

creating the memory image

- when the `loader` runs, it creates the memory image
- guided by the segment header table in the executable
- it copies chunks of the executable
  into the *code* and *data* segments

jumping to the entry point

- after copying the executable, the `loader` jumps to the program's entry point the address of the `_start` symbol
- the start-up code at the `_start` address is defined in the object file `crt1.o` and is the same for all C programs

the crt1.o startup routine

- ```
  0x080480c0 <_start>                 // entry point in .text
      call __libc_init_first          // startup code in .text
      call _init                      // startup code in .init
      call atexit                     // startup code in .text
      call main                       // application main routine
      call _exit                      // returns control to OS
  ```
  ```

Startup code (1)

- *after* calling initialization routines
  from the `.text` and `.init` sections
  the startup code calls the `atexit` routine
- the `atexit` routine registers
  a list of routines to be called
  when the application (`main`)
  calls the `exit` function
- the `exit` function runs
  those functions registered by `atexit`
  then returns control to the os
  by callying `_exit`

Startup code (2)

- when the startup code calls the application's `main` routine, the C code begins to execute
- after the application returns (`exit` is called), the startup code calls the `_exit` routine, which returns control to the os

forking child process

- each program runs in the context of a process with its own *virtual address space*
- the parent shell process forks a child process that is a *duplicate* of the parent
- the child process invokes the `loader` via `execve` system call
- the `loader` deletes the child's initial *virtual memory segments* that are copied from the parent process and creates a new set of *code*, *data*, *heap*, and *stack* segments

invoking the loader

- the new stack and heap segments are initialized to zero
- the new code and data segments are initilialized to the contents of the executable file by mapping pages in the virtual address space to page-sized chunks of the executable file
- finally the `loader` jumps to the `_start` address which eventually calls the application's `main` routine

deferring copying

- during the loading process,
  there is no copying of data from disk to memory
  except some header information

- the copying is deferred until the CPU references
  a mapped virtual page, at which point the os
  automatically transfers the page from disk to memory
  during it's paging mechanism

execve

- ```
  #include <unistd.h>

  int execve( const char *filename,~
                    char *const argv[],~
                    char *const envp[] );
  ```
  ```

execve example

- ```
  #include <unistd.h>
  #include <stdio.h>

  int main(void)
  {
      char *argv[] = { "/bin/sh", "-c", "env", 0 };
      char *envp[] =
      {   "HOME=/",
          "PATH=/bin:/usr/bin",
          "TZ=UTC0",
          "USER=beelzebub",
          "LOGNAME=tarzan",
          0                     };
      execve(argv[0], &argv[0], envp);
      fprintf(stderr, "Oops!\n");
      return -1;
  }
  ```

  https://stackoverflow.com/questions/7656549/
  understanding-requirements-for-execve-and-setting-environment-vars
  ```

.data section

- the .data section contains variables
- Variables change at run time
- the variables need to be in RAM
- Flash, unlike RAM, is not easily changed at run time.
- the flash contains the initial values of the variables in the .data section.
- the startup code copies the .data section from flash to RAM to initialize the run-time variables in RAM.

https://stackoverflow.com/questions/41365110/why-need-linker-script-and-startup-c

linker script

- the object code created by your compiler has not been located into the microcontroller's memory map.
- the linker will do this task and that is why you need a linker script
- the linker script is input to the linker and provides some commands on the location and extent of the system's memory.

https://stackoverflow.com/questions/41365110/why-need-linker-script-and-startup-c

startup code

- a C program that begins at main does not run in a vacuum but makes some assumptions about the environment
- assumes that some variables are already initialized
- the startup code is necessary to put in place all the things that are assumed to be in place when main executes (the "run-time environment").
- The stack pointer
- the constructors of static objects in C++

https://stackoverflow.com/questions/41365110/why-need-linker-script-and-startup-c

load address and run-time address (1)

- When you load a program on an operating system your .data section basically non-zero globals are loaded from the "binary" into the right offset in memory, so that when your program starts those memory locations that represent your variables have those values.

https://stackoverflow.com/questions/41365110/why-need-linker-script-and-startup-c

load address and run-time address (2)

```
unsigned int x=5;
unsigned int y;
```

- in the above code, you expect x to be 5 when you first start
- if are booting from flash, bare metal,
  you dont have an operating system to copy that value into ram,
  it has to be copied manualy.
- all of the .data stuff has to be in flash,
  that number 5 has to be somewhere in flash
  so that it can be copied to RAM.
- So you need a flash address for it
  and a ram address for it.

https://stackoverflow.com/questions/41365110/why-need-linker-script-and-startup-c

startup code

- any function can call any other function
- a local variable x to be 5 and y will be assumed to be zero
- the startup code at a minimum for generic C sets up
  - the stack pointer
  - local variables
  - .bss to zero
  - initialize variables
- if you dont have an operating system
  then you have to code the above
  cannot use system calls (printf, fopen, . . .)
  but depending on toolchain,
  you don't have to write the linker script nor the bootstrap

https://stackoverflow.com/questions/41365110/why-need-linker-script-and-startup-c

linker scripts

- the linker script defines the memory layout of your target and application.
- in the bare-metal programming, there is no OS to handle that for you.
- the start-up code is required to at least set an initial stack-pointer, initialise static data, and jump to main.
- On an embedded system it is also necessary to initialise various hardware such as the PLL, memory controllers etc.

https://stackoverflow.com/questions/41365110/why-need-linker-script-and-startup-c