Link 5. Loading

Young W. Lim

2018-10-02 Tue

Outline

1. Linking - 5. Loading
   - Based on
   - Loading

Based on

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

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

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

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

child process forked

- 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

the loader is invoked

- 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

the copying is deferred

- 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
  ```