CSE 410: Systems Programming

Process Anatomy

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

Last Time

- Structures and structure pointers
- Dynamic memory allocation
- Integer representation
- Floating point numbers

Processes

What is a process?
From the text:

> *[A] process is an instance of a program in execution.*

If a program is a set of machine instructions, a process is:

- Those instructions
- The memory they use
- The system resources they access
- …

Programs

The program that a process runs is loaded from an executable.

An executable is an object file intended to be loaded into a process.

Once loaded, the system provides an execution environment.

UNIX Processes

A UNIX process is protected from other processes:

- It has its own memory.
- It appears to execute on a dedicated CPU.
- The system services it uses are dedicated to it.

Hardware assistance is required to maintain this environment.

This Lecture

In this lecture, we will look at:

- Program (executable) structure
- Memory layout

Executable Formats

Each executable is stored in an executable format.

An executable format provides:

- Information about the environment the program requires
- The program code itself
- Other metadata

Early executables were essentially raw memory dumps.

Such dumps are simply copied into memory and executed.

Modern executables are somewhat more complicated.

ELF

Many modern systems use ELF: Executable and Linking Format.

(Windows uses PE; macOS uses Mach-O.)

ELF executables and libraries contain two types of information:

- Information required to load and execute the object
- Information required to link the object

ELF objects correspond to translation units.

- They provide only linking information

Linking

Linking is the process of creating an executable or shared library from multiple object files.

The linker in the C compiler toolchain performs this task.

It involves:

- Cataloging symbols provided by various object files
- Cataloging symbols provided by external libraries
- Identifying symbols required by objects and libraries
- Binding provided symbols to required symbols

Loading

Loading is the process of moving an executable or shared library into memory for execution.

On Linux, the kernel begins loading, and `ld-linux.so` finishes it.

- The kernel moves various portions of the ELF executable into place
- The kernel moves the loader into place
- The kernel invokes the loader, which performs various changes to the in-memory program data
- The loader jumps to the start of the program

ELF Structure

The data in an ELF file is mapped into sections and segments.

Sections describe the file for the linker.

Segments describe the file for the loader.

The two views typically have significant correspondence.

We will think about an ELF executable in terms of sections.

ELF Sections

Each part of a process is represented in some section.

There are many possible sections, but we will consider:

- Text[1] : The actual program code, as executed by the processor. ELF calls this `.text`.
- Data: Non-code data that has some value defined at compile time; for example: strings, constants, some global variables, *etc.* ELF calls this `.data`.
- BSS: The "block started by segment". This is non-code data that has no value defined at compile time. For example, declared global variables with no initializer. ELF calls this `.bss`.

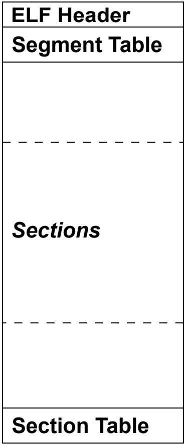
[1] Text, data, and BSS are all historical names.

ELF Object Layout

The ELF header describes the type of object (platform, endianness, *etc.*)

The segment table tells the loader where the parts of the file should be placed in memory.

The section table describes the sections for the linker.



From Program to Process

The executable file is loaded into memory to become a process.

The memory layout of the process mimics the ELF sections.

The system ascribes additional semantics to the loaded layout.

Most POSIX systems will use the same (or similar) layout.

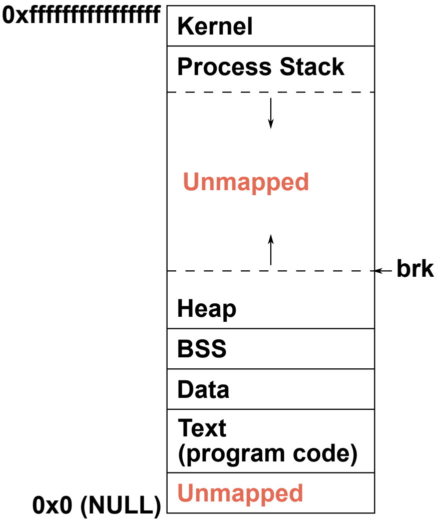
Basic Layout

The lowest addresses are not used — specifically so that `NULL` remains invalid!

The text and data sections come directly from the ELF file.

The BSS doesn't actually appear in the file!

The stack and heap are set up by the loader.



Layout Caveats

As your text points out, modern systems also:

- map shared libraries, which are code used by a program that do not appear in its ELF file
- randomize the location of the mapped sections

However, the logical layout remains the same.

In particular, the order of the sections is maintained.

The Text Section

The text section contains the actual program instructions.

The assembler emits binary machine instructions that are placed in the ELF `.text` section by the linker.

The kernel copies the text into the process's memory, and the loader prepares it for execution by modifying various memory locations.

Data & BSS

The data segment contains variables and constants that have known initial values at compile time.

The linker inserts this data into the ELF `.data` section and the kernel loads it into the process's memory.

The BSS contains variables that have no value at compile time.

The compiler identifies variables in the BSS and records their locations, but does not store them in the ELF image.

The kernel makes space for the BSS when it loads the program.

The Stack and the Heap

The stack contains local variables for function calls.

The heap contains explicitly allocated memory.

Both the stack and the heap can grow.

Thus the unmapped space between the heap and stack.

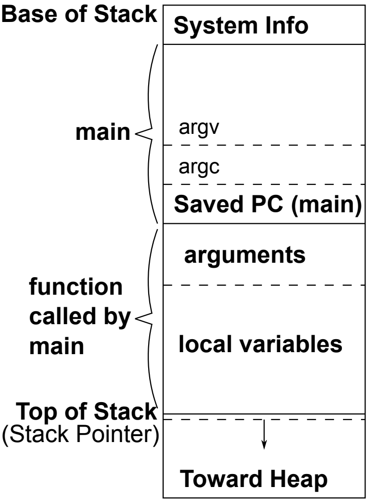
The Stack

The stack grows *downward* as functions are called and shrinks when they return.

The kernel manages the size of the stack automatically.

Each function called has a stack frame that contains:

- The arguments to the function
- Local variables



The Heap

The heap does not grow automatically.

The kernel maintains a program break between the process's data and the unmapped memory between the data and heap.

The program can request that the program break be moved.

Moving the break toward the stack makes more heap space.

Summary

- A program is code that can be executed, a process is that code running on a system.
- The linker joins multiple objects into an executable.
- A loader prepares a program that has been copied into memory for execution.
- Program code (text), initalized data (data), and uninitialized data (bss) are present in both a program and a process.
- The heap and stack can both grow, the former "upward" toward higher addresses and the latter "downward" toward lower addresses.

Next Time ...

- Process creation
- Kernel services
- More about the execution environment

References I

Required Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 7: Intro, 7.1-7.5. Pearson, 2016.

Optional Readings

[2] John R. Levine. *Linkers & Loaders*. Chapter 3: 3.1, 3.7. Morgan Kaufmann Publishers, 2000.

License