Bachelor Informatica

UvA Universiteit van Amsterdam

Dynamic program loading in a shared address space

Leendert van Duijn

May 29, 2012

Supervisor(s): Raphael 'Kena' Poss (UvA)

Signed: Raphael 'Kena' Poss (UvA)

Abstract

In this abstract the research will be summarized to some extent.

Contents

CHAPTER 1

Introduction

When a computer loads a program for execution it is granted its own private address space. In this research we hope to show that more than just a single program can cooperate in a shared address space. This opens up possibilities for processes designers to share hardware between an increasingly larger number of cores without limiting the number of distinct programs.

CHAPTER 2

Theoretical background

2.1 The platform

For this research the Microgrid platform will be utilized. It is . a simulation platform for a massively parallel computation, consisting of multiple general purpose processors based on the DEC Alpha processor architecture. This platform is designed to explore the possibilities of increasingly parallel computing systems focusing on a set [1] of general purpose processing units capable of cooperation at very low cost and latency. Which is hoped to increase trough put.

2.1.1 Features

The Alpha processor

- 64bits address space
- General purpose CPU design
- Risc instruction set

The Microgrid environment has several key features which have led to this research specifically.

- Parallel computation
- Large address space and memory pool
- The entire chip, consisting of multiple cores and caches, shares a TLB
- Houses more parallel processing power than a typical application needs...

In order to facilitate communication between the processing cores and on chip components such as the Memory Management Unit the Microgrid processors have an on chip peer to peer network dedicated to small packets. This network is depicted in Figure 2.1.1

Figure 2.1: On Chip Network
Courtesy of Raphael 'Kena' Poss.

[1] Documentation about: Slr defaults to: 128-core, ML's COMA, 1GHz cores, 4xDDR3-1600, 128K L2 caches

2.1.2 The Problem

The Microgrid platform offers a loader capable of loading a single program which then gains full control of the system. The Microgrid offers a virtual memory system to programmer where a single memory management unit is shared across the chip which holds more than one core where each core has at least one cache. This would mean that either any memory access goes through the memory management unit before it can be processed, even if the memory being requested is in the cache of the current core. In order to achieve performance the caches could use virtual addresses instead of physical addresses, this however introduces collisions as more than a single program would normally share a cache. For programs this effectively means either sharing the address space regardless from the main memory layout. Or the anti architectural limitation of one program per cache, where any shared cache would have each and every request go to the memory management unit first which would lead to differing impact on the overhead depending upon the cache level being shared.

2.1.3 The Solution

The proposed solution for this problem is sharing a single address space across several programs which would eliminate the need for the blocking call to the memory management unit. This solution has a significant drawback, permissions based on address space isolation will be void. In order to achieve permissions per process a call could be made to a component such as the Memory Management Unit, which holds a capability tables for PID to permission mapping. These calls would be issued in parallel to the cache access without any delays if the permission is granted and cache invalidation on a denied.

The goal of this research is realizing a loader capable of settings up the environment in such a way any set of programs can be run without colliding while they share the single address space, so that consulting the Memory Management Unit is only done for requests to the main memory, and possibly non blocking permission checks.

2.2 The loading

Loading a program has several phases to go through. At the end of these steps a traditional loader would transfer control to the loaded software.

2.2.1 User input

The loader needs to be able to adept itself to the needs of the user, most commonly a user will want to tell the loader what programs need to be loaded and what specific parameters or settings should be used. This can done by means of a simple configuration file as detailed in Section 2.3.2.

2.2.2 Loading from ELF

The loader will need to load the program the user has requested, program code can be packed and stored in many file formats. The ELF file format has been chosen for this loader as supports some key features needed for our loader such as Relocation information and the Dynamic Symbol table, these features are primarily needed for relocation. The preexisting loader for the Microgrid platform has been used as a stepping stone, as it is already capable of loading an ELF file in a straigtforward way to the Microgird platform..

The loading of an ELF executable proceeds generally as specified in [1]. Loading would result in a set of memory ranges being populated with code and data. Administrative features included in the ELF format like the program entry point and symbol data are used beyond this point though they are not necessarily part of a fully loaded program.

2.2.3 Location decisions

Since programs will share the address space in a parallel fashion a single arbiter needs to decide where a process can be loaded. This is due to the risk of independent programs memory content overlapping each other. Such overlap if unintentional could lead to massive memory corruption.

The arbiter is a sequential component in an otherwise parallel system[2]. In order to retain high performance with an increasing amount of processes a freelist could be maintained. This list points to an administrative entry which is guaranteed to be either available or a truthful indicator that there is no space whatsoever. It contains a link to the next available entry. On process termination, its entry can be attached to the front of the freelist ensuring that all memory ranges are accounted for at any time.

The freelist maintains constant complexity over an increasing amount of processes in the system, it does however demand locking/serialisation of the requests. To achieve this locking the loader runs all functions which concern the freelist information on a core demanding the thread runs without any interruption from intervening threads, this is done via the `sl_exclusive` flag for the `sl_create` command.

2.2.4 Relocation

During the loading process an exact location is determined for the program. This location is highly dynamic as it depends on the current memory occupation and deallocation history. When the program is requested to load any other program load can affect its final location. This introduces the need for program relocation [3]. The relocation can be split into two important phases. The code relocation and data relocation. The code is not always trivially relocated [4]. To protect the scope of this research the loader demands for the loaded programs to be compiled with several flags related to Position Independent Codeso that the code is functionally independent of its location in memory. The needed flags are elaborated on in Section 3.5.3. This leaves some data relocation entries to be processed by the loader in order to correct data pointers[5] These relocation corrections are done as specified in [6]. This is summarized as adding the programs base address to each pointer the compiler has flagged for correction. These pointers are full size pointers which places no extra limitations on program location. Some of these pointers could be function pointers for usage by the Position Independent Codethis is however irrelevant to the relocation code.

2.2.5 Process private memory

As programs may require arguments and environment variables which outlast the parent process they require storage allocation. This allocated room is not required for small or specialized programs who will not use them, as such this is optional.

This allocation is supported by the ELF file format by including a special section which reserves space for either arguments, environment or other custom data segments. This section is detected during loading and if present will be used to pass any argument and environment variables. If this section is absent no arguments will be passed to the program. This enables a minor speedup and memory saving for programs which are known not to use arguments[7].

2.2.6 Execution

Transferring control to the loaded program. This is done via the Microgrid function sl_create which places the program on the desired cores, taking as parameters the address to start execution at and any arguments to pass. It offers an option to fully reserve the cores for the given program, blocking any core sharing. Due to the serializing effects of this option and the absence of truly intelligent core selection this is disabled by default.

---

[2] this process could be paralleled to some extend by dividing the possibilities over a set of arbiters and choosing the earliest available arbiter, this introduces overhead and does not solve the need for a single (arbiter) arbiter

[3] Documentation about: position dependent code

[4] Documentation about: coderelocationhard

[5] For an example and explanation see Section A.1.1.

[6] Documentation about: elf reloc data reference

[7] Documentation about: example no arg programs and speedup

2.3 User control

The loader can be influenced by a user in several ways. During its preparation and compilation several settings can be tweaked for optimum performance as will be specified in Section 2.3.1. After the loader has been compiled and linked the remaining tweaks need to be done via configuration, most tweaks are program specific so that once a program is loaded other ill-written programs can not legally affect it[8].

2.3.1 Preparation and Compilation

The performance of the loader is influenced in many ways such as the macro definitions, configuration settings and the loaded programs. During compilation some values such as the memory size for programs can be tweaked. The default cores for some operations... debugging print...

- Ranges Base
- Ranges Size
- Maximum number of programs
- Core used for serialized printing
- Core used for administrative functions

2.3.2 Configuration details

The user can guide the loader by using a quite powerful configuration file. The file can be easily written in any text editor capable of saving as plain formatted text. Some obligatory settings

- Filename of the ELF file (string)
- Arguments, can be an empty line (newline separated strings)
- Environment, can be an empty line (newline separated strings)

Some settings are optional but would be present in most cases

- Verbose, "true" or a numerical value"
- Exclusive, "true" or "false" (Optional, default false, TODO)
- Core_start, numerical core number, (Optional)
- Core_size, numerical number of cores, (Optional, defaults to 1)

[8]Some efforts of sabotage are predicted to be effective as detailed in [9]

CHAPTER 3

Implementation

3.1 Assumptions and constraints

In the implementation process of the loader several assumptions had to be made, most of which will be demonstrated to be correct.

- Availability of a working C compiler, gcc
- Availability of a working linker, gcc
- ELF file format output for the linker
- Availability of the -fpic -fPIC and -shared flags
- Correctness of loaded code
- Relocateability of loaded code
- Loaded code will not try to harm other loaded programs

3.1.1 C compiler

The C compiler used during development is the SLC [1]compiler which has shown to compile an extensive set of test programs such as the benchmark set…

3.1.2 The linker

In order to link the loader, which is divided over a functionally distinct set of source and header files. SLCis used which is fully compatible with the used C compiler. The compiler is primarily tasked with generating object files from the loader source. The linker for linking these together in a self contained executable. Optionally including relocation information if runtime relocation of the loader if desired.

3.1.3 Flags for linker and compiler

For the loadable programs some extra restraints exist, the C compiler and linker need to follow some specific rules in order to maintain full relocatability and functional correctness. These flags are the -fpic -fPIC and -shared flags. The -f flags indicate to the compiler that any executable code needs to be fully relocateable. The -shared flags indicated to the linker all data references

[1]Documentation about: `slc 3.7b.28-dac6`
`Copyright (C) 2009,2010 The SL project. This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.`
`Written by Raphael 'kena' Poss.`

might be relocated prior to execution and as such administration to support should be included. These flags are needed to compile any program that should reliably run within our loader.

To implement the loader the C programming language is employed, a compiler converts this into executable code for the Microgrid platform.

3.2 Api

The loaded programs currently lack a full libc, as the runtime which is used to link and locate the library is making false assumptions about the system and its no longer private address space. In order to offer the loaded applications some of the missing functionality and more importantly access to several loader related functions an api structure is defined, which holds pointers to functions as offered by the loader. The loader passes the pointer to all loaded programs which can be used to accessed loader functions, this is done by passing the pointer to the single struct as a parameter in a register.. This pointer points to a struct containing function pointers for these functions:

- spawn, spawn a program
- print_string, prints in an orderly fashion (blocking)
- print_int, prints an integer in an orderly fashion (blocking)
- print_pointer, prints a pointer in an orderly fashion (blocking)
- load_fromconf, loads a program from config file
- load_load_fromconf_fd, loads a program from config from file descriptor
- load_fromparam, loads from structure with parameters
- breakpoint, TODO loader break point for single program

3.3 Platform dependency

We depend on the microgrid for several key functions, these would need to be replaced if any other platform where to be targeted.

- sl_create, for creating the program stack and core allocation
- sl_detach, letting the loader detach from a loaded program
- mgsim_control, for memory range allocation and deallocation

3.4 Configuration

The loader accepts a configuration file which contains everything it needs to know in order to prepare, call and clean up a loaded A simple scanner which parses keyword value pairs, these are then terminated by a blank line at which point the arguments can be specified. These arguments will be passed as the traditionally called argv, which can only be done if the necessary room is reserved in a section as detailed in Section 3.5.1. These newline separated arguments are terminated by a blank line. After this blank line the environment variables are once written separated by newlines. The environment variables should be in the form `a=b`. The environment variables are terminated by a blank line after which any remaining data would be left untouched.

3.5 Elf loading

3.5.1 Special symbols

The loader searches for some special symbols which it can use to store and pass arguments to programs in an unobtrusive manner. These symbols are generated by including a C source file during compilation[2] of the loadable program. These are symbols with global scope, which is global to the compiled program. Other loaded programs do not see them. These symbols are detected when parsing the dynamic symbol table and include both the size and the unrelocated location, after correcting for relocation the symbol location is stored in the programs administration for later use. The symbols are recognized by their names, these can be changed by altering the definition of `ROOM_ARGV` or `ROOM_ENV` in the file `loader_api.h` and the related C source file which would be either `argroom.c` or `envroom.c`. The latter also permit the size to be modified in order to accommodate for the anticipated amount of arguments.

Size constraints and guidelines

The size the argument and environment objects require depends on the anticipated input, in order to calculate the most efficient size these formula should be used:

$$Size_{Env} = 1 + \sum_{i \in environment} (1 + strlen(i))$$

$$Size_{Args} = 8 * (Argc + 1) + \sum_{i \in argv} (1 + strlen(i))$$

The room needed for the arguments considers the storage for the argv array, the environment room does so for the final null byte. These storage locations are only related in concept and implementation. They are fully independent so one may choose to include any combination of sizes.

In the situation insufficient room is available for the arguments the loader will print a warning message, setup to pass no arguments whatsoever. It will then check the same for the environment variables. It will still try to execute the program even if these checks both fail by passing null pointers and an argc of zero to indicate no arguments could be passed. It is left up to the developer of the loaded program to decide whether it can successfully execute in their absence.

3.5.2 Algorithm

The Elf file is read into memory where a simple algorithm is followed.

```
Load the file into memory
Inspect the header
Locate the program headers
Scan the program headers for the base address
Find an available PID
Determine the read base address, aligned
Loop over the program headers:
  Load segments to their destination
  Zero leftover memory
Locate the section headers
Scan the section headers for the Dynamic Symbol table and Relocation tables
Scan the Dynamic Symbol table for special symbols:
  Note the location and size for the argument and environment room
For all found Relocation tables:
  Loop over all entries in the table:
    Determine the pointer location
    Determine the symbol and offset
```

[2] linking an object file compiled from this file will achieve the same effect

```
    Calculate and update the pointer
Prepare the arguments and environment if room is available
Spawn a thread which will call the main function as found in the entry point
```

The loader optionally includes a verbose set of print statements useful for debugging purposes. This can be disabled for performance reasons by changing a macro definition or disabled at runtime by passing a verbosity setting to the loader.

The loader is guided by a configuration file which describes what program should be called with optional arguments and program specific settings. This configuration file is covered in detail in Section 2.3.2. The loader follows two simple algorithms for parsing.

```
Reading key value pairs:
Start:
  Key=""
  Value=""
  Buffer=""
  Read character X:
    If X == '=':
      Key=Buffer
    If X === '\n':
      if Key == "":
        Goto Done
      Value=Buffer
      Goto ParseSetting(Key, Value)
    Buffer += X
    Goto Read character X

  ParseSetting(Key, Value):
    Pick the setting based on Key, set it using Value
    Return

  Done:
    Finish up, settings done
```

At this point all settings have been parsed, the programs filename is known and the settings have been terminated with an empty line. At this point the command line arguments can be set.

```
Argc=1
Argv[0]=ElfFilename
Read character X:
  if X == '\n':
    if Argv[Argc][0] == '\0':
      Goto Done
    Argc++
  Argv[Argc] += X
  Goto Read character X
Done:
  Finish up, Arguments known in Argv and Argc
```

The same is the done for the Environment substituting Argc and Argv for EnvC and Evnp.

3.5.3 Program limitations and requirements

Some compilation flags and settings are explicitly required in order to reliably load a program.

- -fPIC
- -fpic

- -shared
- crt_fun.c
- -nostdlib

These flags tell SLC [3]to compile position independent code, to include data relocation information and replace the C runtime codewith a bare one which consists of a wrapper which calls the main function of the loadable program.

3.6 Spawning an initial program

The loader initial program is loaded by passing arguments which will be parsed as configuration files, loading them in sequence on either the default core or the specified cores. These files should adhere to the format as specified in Section 2.3.2. Several examples are included in Section A.2

The loader in this research will behave in ways like loaders normally found in userspace within an operating system environment. As such it will not offer full control of the system as a bootloader would, it will run in userspace and it resides in virtual memory. It is however designed for a system lacking a full operating system, it therefore currently lacks some features that most operating systems offer through the loader. The most prominent missing features include program exception handling, a loaded program which performs illegal operations is likely to terminate the entire loader and all loaded programs. Library support, programs currently lack a way to share libraries dynamically which could mean increasing redundancy as more and more statically linked programs include their own copy of common code. The loader will not treat debugging information in any special way, and as such might require expansion if a debugger is introduced to the system.

3.7 In program Loader calls

In order to allow more complex program structure we offer programs an API through which they can invoke loader functions to spawn programs or perform other related tasks.

3.8 Implementation considerations and reflection

A perfect design is rare, as not all optimum settings are fully compatible.

3.8.1 Location dilemma

Size

A program needs a location which can not be easily changed at run time. The run time of a program may be unbounded and as such a single program introduces an obstacle in the memory space. This is largely an allocation problem where prior to execution exact space requirements may not be available.

Location

When loading a program care has to be taken to ensure no programs are given overlapping address spaces[4].

---

[3]Documentation about: `slc 3.7b.28-dac6`

`Copyright (C) 2009,2010 The SL project. This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.`

`Written by Raphael 'kena' Poss.`

[4]All programs share the same address space, but to their knowledge the subspace they inhabit is a traditional address space

During the debugging of any collection of programs one would like to know which program is responsible for certain instructions, problems or memory usage. In order to trace a specific memory location to a program we would need some sort of standard procedure.

Solution

As our loader is designed with a 64bits address space in mind we have adopted a formula for base address calculation in which a process is given a base address based on its identifier and a predetermined size. This size is the upper limit for any loaded process sub address space. This size can be changed prior to loader compilation.

$Base = Base_{Global} + (Id_{Process} * Size_{maximumsubspace})$

This enables us to efficiently determine a base address for a process and also pinpoint the source of many memory related problems as an offending instruction can be trivially traced back to its program based on its address.

3.8.2 Trace ability of problems

As our loader loads an increasingly large number of programs problems memory ownership needs to stay intuitive. In order to trace errors we can determine ownership by calculating the Id of the memory based on the inverse of our location formula.

3.8.3 Relocation

The loaded program is loaded to an a priory unknown location so the loader has to finish the relocation process. This is done by using relocation information information stored in the section headers. The loader parses the section data. [2]

CHAPTER 4

Progress report

During my research I have reached several conclusions.

4.1 Milestones

4.1.1 Planned milestones

A loader for single program.
A loader for multiple programs.
A loader with in program spawn function.
A loader with Input and Output redirection.

4.1.2 Unexpected roadblocks

Missing functions.
Libc conflicts, loading an existing libc leads to crashes.
Runtime cleanup, programs not being cleaned due to thread termination.

4.1.3 Reached milestones

Loading a program.
Loading more programs.
Loading based on user configuration.
Loading on a specific core.
Letting programs output in a orderly fashion.

Loading something

A single program being loaded, though it seems trivial it is quite the relief when it finally does.

Loading several programs

The more significant milestone, loading multiple programs which execute as they would in a private address space. With the significant difference that they share their address space between themselves and the loader. The second large milestone reached is the usage of global variables which due to their relocation needs was a hassle to get right.

4.2 Future research

4.3 Security

As the loader is focused on sharing an address space between programs some assumptions where made as seen in [1]. One of these assumptions is the willingness to cooperate. Each program loaded has the same rights as any other to the same memory without the isolation most application programmers are used to

...What has been found to be unanswered for now...What could be better...

[1] Documentation about: assumtionsprograms

CHAPTER 5

Experiments

5.1 Testing

During the development several programs where written to test nominal behavior. These programs are designed to make use of several features the ELF file format which could break on loader malfunction.

5.1.1 Relocation

The `tinyex.c` prints strings which are globally defined in an array. This array of string pointers requires runtime relocation to ensure they point to the relocated string data. These are full size absolute pointers and as such corrected by simple addition of the program base offset which is unknown at compile time.

Symptoms of malfunction for this program would include illegal memory access and the attempted printing of non string data.

5.2 Limitations and future work

5.2.1 Permissions

In order to explore possible problems nasty programs were constructed. These have been used during testing to improve the loader. There is however another class of programs, malicious programs which attempt to access memory which was allocated for another loaded program or even the loader itself. Due to the lack of memory protection methods beyond the normal read write and execute permissions all programs share these permissions. As such any program could take control of most other programs.

A means of protecting loaded programs from each other is documented in [3]. Their proposed protection system would enable fine grain access control and secure the loader and programs from ill written programs if not malicious programs.

5.2.2 Library sharing

The loader could be extended to dynmicly load libraries in such a manner that multiple programs can share them in existing operating systems this has shown to decrease program sizes and reduce memory needs.

5.3 Applications

The loader offers a platform which could be extended to allow dynamic task execution and placement. A shell program could be used to offer a dynamic interface. Combined with other programs and daemons a simple operating system could be realized.

CHAPTER 6

Conclusions

6.1 Benchmarking results

In benchmarks we have seen that performance. . . It quite likely works

6.2 Stability

The loader lacks some of the protection mechanisms required for the stable execution of untrusted code. However under normal execution the loader is quite stable and handles any errors it can by terminating the offending program and offering several handles for debugging purposes.

During testing ??? programs have been run in tandem. ??? programs have been run on the same core. ??? programs have been run in a recursive manner.

6.3 Final conclusion

The system. . .

6.4 References

Bibliography

[1] Tool Interface Standard (TIS) Committee. *Executable and Linking Format (ELF) Specification, Version 1.2*, May 1995.

[2] John D. Polstra. *FreeBsd Source src/sys/sys/elf64.h, version 1.9, 5 May 2012, http://freebsd.active-venture.com/FreeBSD-srctree/newsrc/sys/elf64.h.html*, September 1999.

[3] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proc. 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, ASPLOS-X, pages 304–316, New York, NY, USA, 2002. ACM.

APPENDIX A

Problems and in depth solutions

A.1 Bugs

A.1.1 String data relocation

At an early stage in the loaders development progress all data relocation was done at compile time. The assumption was made the compiler would generate code to correct data pointers included in initialized variables. However this is not the case as was concluded when a simple program designed to print an array of strings was run and it became clear that these strings where assumed to be at a fixed location. The error in the location could be expressed as the loaded programs base. The programs continued to show this behavior when compiled with the -fPIC and -fpic compiler flags.

In order to solve this problem, which is a symptom of an incomplete relocation process the data pointers need to be corrected. In order to know which data needs to be corrected for the actual base the compiler needs to be told that the loader will finish the loading process. This is done by passing it the -shared flag. This flag prevents the compiler from incorrectly assuming a value for the definitive base address and include relocation information into the produced ELF file.

Example of the printy programs reported sections when not compiled with -shared

```
Section   #(    Type):             Name, Type#          Flags,          Addr,           Off,
Section   0(NULLTYPE):                 ,     0,              0,             0,             0,
Section   1(Progbits):            .text,     1,              6,      16777216,         65536,
Section   2(Progbits):          .rodata,     1,              2,      16778048,         66368,
Section   3(Progbits):    .eh_frame_hdr,     1,              2,      16778112,         66432,
Section   4(Progbits):        .eh_frame,     1,              2,      16778136,         66456,
Section   5(Progbits):             .got,     1,              3,      16843720,         66504,
Section   6(  Nobits):             .bss,     8,              3,      16843720,         66504,
Section   7(Progbits):         .comment,     1,              0,             0,         66504,
Section   8(Progbits):         .argroom,     1,              0,             0,         66522,
Found the argument section
Section   9(  Strtab):        .shstrtab,     3,              0,             0,         74714,
Section  10(  Symtab):          .symtab,     2,              0,             0,         75576,
Section  11(  Strtab):          .strtab,     3,              0,             0,         75984,
Spawning program from 0x80101230 of size 0x1290e with flags 2
To cores: 1 @ 0
Returning from Loader main
```

The same program With -shared

```
Section   #(    Type):             Name, Type#          Flags,          Addr,           Off,
Section   0(NULLTYPE):                 ,     0,              0,             0,             0,
```

```
Section   1(Progbits):           .text,     1,              6,      16777216,          65536,
Section   2(    Hash):           .hash,     5,              2,           400,            400,
Section   3(  Dynsym):         .dynsym,    11,              2,           576,            576,
Section   4(  Strtab):         .dynstr,     3,              2,           792,            792,
Section   5(    Rela):       .rela.plt,     4,              2,           848,            848,
Section   6(Progbits):         .rodata,     1,              2,      16778048,          66368,
Section   7(Progbits):   .eh_frame_hdr,     1,              2,      16778112,          66432,
Section   8(Progbits):       .eh_frame,     1,              2,      16778136,          66456,
Section   9( Dynamic):        .dynamic,     6,              3,      16843720,          66504,
Section  10(Progbits):            .plt,     1,              7,      16844016,          66800,
Section  11(Progbits):            .got,     1,              3,      16844064,          66848,
Section  12(  Nobits):            .bss,     8,              3,      16844072,          66856,
Section  13(Progbits):        .comment,     1,              0,             0,          66856,
Section  14(Progbits):        .argroom,     1,              0,             0,          66874,
Found the argument section
Section  15(  Strtab):       .shstrtab,     3,              0,             0,          75066,
Section  16(  Symtab):         .symtab,     2,              0,             0,          76352,
Section  17(  Strtab):         .strtab,     3,              0,             0,          76976,
Spawning program from 0x80101230 of size 0x12d27 with flags 2
To cores: 1 @ 0
Returning from Loader main
```

As can be observed, the RELA section.

A.2 Example Configurations

These configurations load a single program each with the specified arguments, environment and specific loader settings.

```
verbose=true
filename=/path/to/file/elffile

argv1
argv2
argv3

env0=1
env1=cookies
env3=needs more ducktape
env4=sudo su
env5=make sandwhich -j9001
```