Bachelor Informatica

Universiteit van Amsterdam

Dynamic program loading in a shared address space

Leendert van Duijn

June 7, 2012

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

Signed: Raphael 'Kena' Poss (UvA)

Abstract

In our research we have implemented a system capable of dynamicly loading programs onto a Microgrid. This loader will allow an enduser to place tasks onto an Microgrid without the need for constant recompilation.

Contents

CHAPTER 1

Introduction

1.1 Context

Modern computing platforms allow the user to load programs which perform some task or computation. These systems typically allow not just a single program to run but enable some form of sharing of the available resources. This is typically done by an Operating System, abbreviated as OS. An OS will typically a collection of functions or even programs which maintains control over the resources where any client or userspace activity is granted only those limited rights and control they need. Users activities take place in processes. Each of these can have access to some randomly accessible memory and limited computing time. Along with file interaction process state is maintained and organized by the OS. The execution of a process it done through threads. A threads is a segment of program code which executed sequentially. Each activity consists of one or more threads where threads may be started at any moment during execution and they are executed parallel to each other. All available[1] threads are commonly executed concurrently with each other and any other threads running on the system.

In a single chip, single core processer setup the OS would typically interlace thread execution with those of that of any other thread the user has initiated. This is a time sharing system where all seemingly parallel execution needs support from the OS. In a multi chip or multi core design this restriction is lifted to some extent, the architecture now allows true parallel computation although in most traditional systems the number of available cores, whether on a single chip or a local network of chips, is outnumbered by the amount of running processes. A trivial example would be the laptop this document was written on, it has 2 processing cores and is typically running around a hundred processes. Most operating systems still interlace the execution of all the processes as they would in a single core environment but with the added benefit of being able to execute several threads in true parallel.

An example of such multi-core, multi-threaded environment is the Microgrid. The Microgrid environment is a novel platform for the parallel execution of programs with a large number of general purpose processing units capable of OS independent thread management and fast thread creation. This opens up possibilities to make programs massively multithreaded where the traditional overhead is reduced to a minimum.

1.2 Memory architecture and virtual address spaces

This system is controlled and centered on the Memory Management Unit. The platform used in this research has a single Memory Management Unit per chip. This unit is shared across the chip holding more than one core. In Figure 1.1 eigth cores can be seen sharing an L2 cache per four cores, where the MMU is only accessed on DDR memory access.

In traditional multi-core chip architecture which groups together exising core designs previously developed for single-core chips (eg Intel or ARM cores) each core is equipped with its

[1] Some might not be available due to unresolved data dependencies

Figure 1.1: 4 cores sharing L2 cache

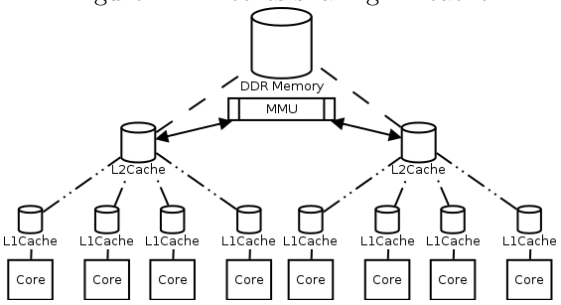
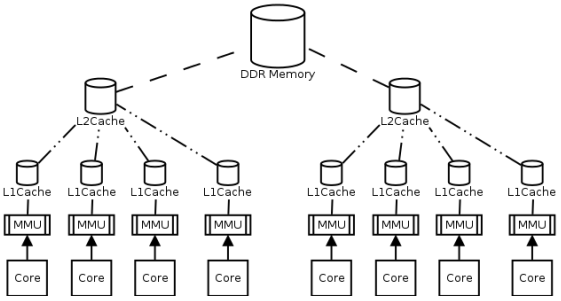


Figure 1.2: 4 cores sharing L2 cache with single MMU



own MMU. In this context it is assumed each process defines its own virtual address space. To prevent homonyms on the cache system the translation of virtual to physical addresses should happen *before* the caches are accessed. This places the MMU *in the way* of memory accesses. As can be seen in Figure4cachemmy. The architecture must optimize MMU access by, for example investing in complex Translation Lookaside Buffer (TLB) structures, to ensure the time to translate stays as short as posible.

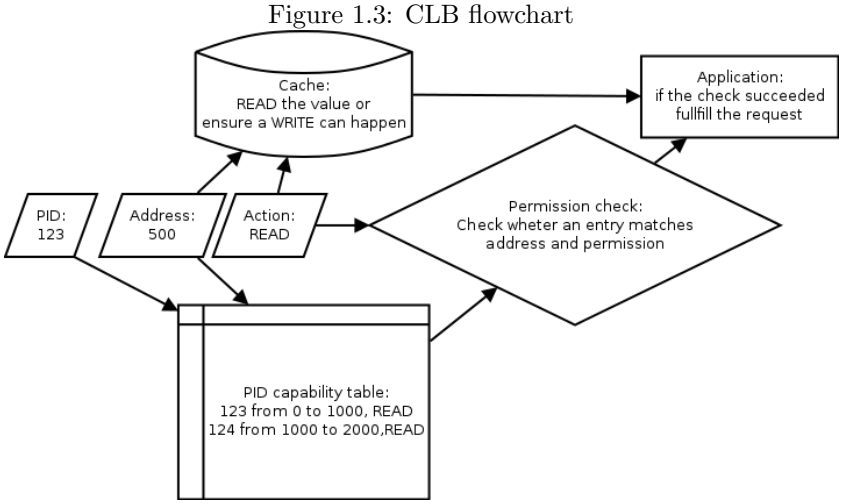
In order to achieve performance the caches could use virtual addresses instead of physical addresses. As a consequence programs would need to have distinct virtual addresses. For programs this effectively means sharing the address space, regardless from the main memory layout.

An alternative solution would be introducing a limitation of one program per shared cache. This would restrict any threads running on a core which has access to the untranslated cache to belong to a single program. Any shared cache or memory access would have go to the memory management unit first which would lead to differing impact on the overhead depending upon the cache level being shared. This would reduce a muli-threaded core to the execution of one single program which contradictive to the intended multi-core multi-threaded, highly parallel design.

1.3 Proposal

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. In contrast

Figure 1.3: CLB flowchart



to most virtual memory systems, isolation based on seperate address spaces cannot be used.

The implementation of access control requires a compononent that checks all memory accesses issued by programs against a table that maps address ranges to access permissions. In traditional systems this check happens in the MMU. The translation/checking step on the path to the L1 cache is accellerated by a Translation Lookaside Buffer (TLB), containing both translation and permission information.

When translation and access control are seperate and the cache system uses virtual addresses in a shared address space, it becomes possible to perform access checks in parallel with the cache access. This can be done e.g. with a Capability Look-aside Buffer which can report wheter an access is allowed at the end of the L1 cacche access. Thus increasing and opportunity for parallelism.

In Figure 1.3 a flowchart can seen for an access to a shared cache.

1.4 Contribution

The goal of our research is to demonstrate the feasibillity and benefits of a single virtual address space shared by multiple independent program, such as the one provided the Microgrid architecture. To achieve this, we implement the components of an operating sustem for the Microgrid in charge of loading and starting programs in a shared address space. Our proposed components include:

- A memory manager which divides a 64-bit virtual address space in large regions and interacts with the "virtual memory manager"[2] to allocate and deallocate physical memory.
- A process manager which tracks which memory has been allocated per process, and provide seperate API handles to each program.
- A program loader which is able to relocate ELF data sections over the shared address space on-the-fly and configure the segments access permissions specified by executable files using the hardware MMU.
- Using our technology, we are able to show that multiple programs compiled seperately can be loaded, share the virtual address space and interleace on microthreaded cores without the overhead of switching address space on context switches between threads.

[2]Accelerated in hardware

Prior work

A single shared address space is not unique, OPAL [1] proposes a distributed system sharing a single address space. The Mungi system [3] is another system which shares an address space among all local programs.

CHAPTER 2

Problem analasys and synthesys

2.1 Platform

For this research we will use the the Microgrid platform. The Microgrid is a many-core architecture developed at the CSA group at the University of Amsterdam which combines hardware multithreading on each core and hardware logih to optimize the distribution of program-defined threads to multiple cores[1]. This platform is designed as a research vehicle for the exploitation of fine-grained , massive parrallelism on chip.

In our work we use a software emulator of the Microgrid called MGSim. This emulator implements Microgrids with configurable hardware parameters, such as the number of cores, ISAs and cache sizes. We intend our loader program to be compatible with any Microgrid configuration. However, as a reference configuration we use a Microgrid of 128 cires with each core implementing a 64-bit DEC Alpha ISA.

2.1.1 Features

64bits address space

The virtual memory system has a 64 bits address width. All integer registers have a 64-bits size.

General purpose CPU design

The Alpha processor was designed as a true general purpose unit.

Large address space and memory pool

The large potential amount of memory addresses opens up possibilities to run more memory intensive programs concurrently. Typical computation platforms offer the possibility to run several programs either interleaved or in parallel and the main memory is shared among the many running processes.

Houses a lot of parallel processing power

Microgrids can be configured with a diversity of hardware parameters, such as the number of cores and cache sizes. The default configuration in our project defines

- 128 D-RISC cores with an Alpha ISA, each with 6KB[2] of L1 cache
- 128KB L2 caches, shared by groups of 4 cores
- 4 DDR3-1600 external memory channels

[1] http://svp-home.org/microgrids
[2] 2KB code, 4KB data

The entire chip, consisting of multiple cores and caches, shares a MMU

As a microgrid chip houses many cores with several caches it offers a lot of potential for parallel computation. The chip does however have a single Memory Management Unit. This component is responsible for translating virtual address into physical addresses. In a many core configuration this leads to the general issues detailed in Section 1.2. In Figure 2.1.1 a network of these cores can be seen.

Inter-component communications network

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 component control and configuration, as such it is optmized for low bandwidth and low latency[3]. This in contrast to the memory network which is optimized for high bandwidth at the expense of high latency[4]. The networks are depicted in Figure 2.1.1.

2.2 Overview of the loading problem

2.2.1 Problem statement

The memory manager in our system will need to decide where in the available virtual address range in order for the process to place a loaded programs memory image. In this context the loader will fulfill the task of an operating system. It will decide which regions of virtual memory can be used to load a program and will ensure this memory can be recycled should the process terminate.

This allocation of a memory range and deallocation is bound to several critical sections which prevent it from being fully independent. As such the memory manager needs to ensure several actions are executed in an essentially sequential manor. In order to improve performance and utilize parallel capabilities these sections should not include any code which could be safely executed in parallel.

In order to offer timing instruments the process manager should be equipped with the means to time loaded processes and offer insight in not only the loaded programs performance but also the loading of that program.

2.2.2 User input

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.

The loader needs to fullfill 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. We propose to to do this via a configuration file, although our work can be easily extended to use an API instead. For information on the implemented configuration file see Section 2.3.2.

2.2.3 Loading from ELF

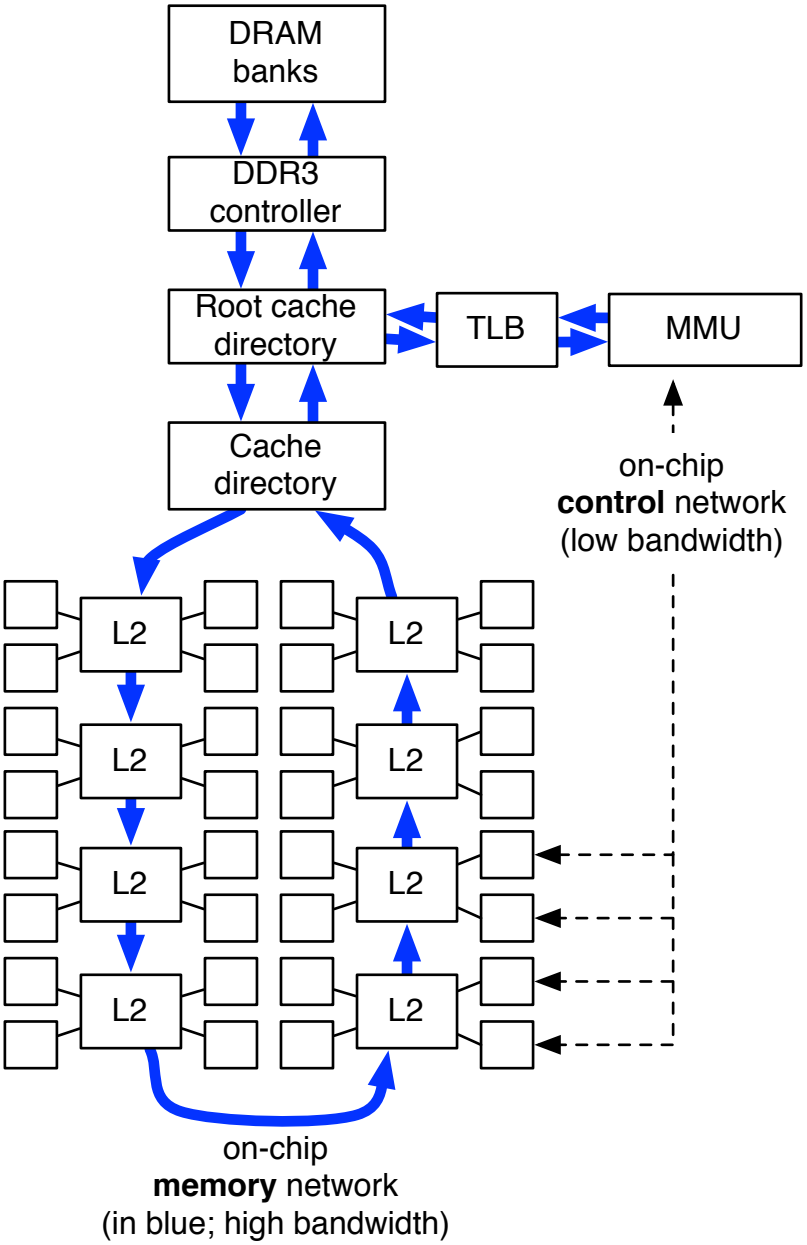
The loader will need to load the program the user has requested. Program code can be packed and stored in many file formats. The ELF file format has been chosen for this loader as supports some key features needed for our loader such as Relocation information and the Dynamic Symbol table, these features are primarily needed for relocation.

There are plenty of implementations for ELF loaders we could have used. We have used the MGSim implementation. which loads a boot ROM into the memory upon system initialization. We used this code as an inspiration for our loader.

The ELF file format is the defacto standard for executable files. The loading of an ELF executable proceeds generally as specified in [2]. Loading would result in a set of memory ranges

[3] 20 pipeline cycles to control across the chip
[4] hunderds of pipeline cycles to move data across the chip

Figure 2.1: A 32 core Microgrid



Courtesy of Raphael 'Kena' Poss.

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.

2.2.4 Location decisions

Since programs will share the address space in a parallel fashion, a single arbiter, our memory manager, needs to decide where a process can be loaded. This is to prevent independent programs memory overlapping each other.

The memory manager is a sequential component in an otherwise parallel system[5]. 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. The entry of a terminating processes 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/serialization of the requests.

2.2.5 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 a program is requested to load, any presently loaded program affects its final location. This introduces the need for program relocation.The relocation can be split into two important phases. The code relocation and data relocation. The code is not always trivially relocated [6]. To limit the scope of this research the loader demands for the loaded programs to be compiled with several flags related to Position Independent Code so 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. For an example and explanation see Section A.1.1. These relocation corrections are done as specified in [7]. This can be 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 Code. Hoever the function pointers themselves are only data and can be considered as such by the relocation code, with no distinction from other data types.

2.2.6 Process private memory

As programs may require arguments and environment variables which outlast the parent process they require their own storage whose lifetime is bound to the loaded process and not the process invoking the loader. This allocated room is not required for programs which do not follow the C convention of passing command line arguments and environment variables, as such this is optional.

We propose to allocate this storage by making the ELF file format include 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 these arguments.

2.2.7 Execution

Our process manager governs the transferring control to the loaded program. Its primary tasks in this context are debugging support, timing control, and most importantly transfering partial

[5]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

[6]Documentation about: coderelocationhard

[7]Documentation about: ELF reloc data reference

control of the system to the fully prepared programs memory image. This is done via the Microgrid construct sl_create which places the program on the desired cores, taking as parameters the address to start execution at and any arguments to pass.

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 behaviour of the system, most specificly the loader, is controlled by both static and dynamic parameters. Static parameters are given by the C preprocessor macros and type definitions in the sourcecode. The dynamic parameters are given via configuration file, described below in Section 2.3.2. The static parameters are:

- base_off in loader_api.h
- base_progmaxsize in loader_api.h
- MAXPROCS in elf.c
- PRINTCORE in basfunc.c
- MEMCORE in basfunc.c
- NODE_BASELOCK in elf.c
- minpagebits in basfunc.c
- maxpagebits in basfunc.c
- Identifier string for argument ELF section, ROOM_ARGV in loader_api.h
- Identifier string for environment ELF section, ROOM_ENV in loader_api.h

The base_off setting allows the tweaking of the base location, this so that the loader may evade certain areas of memory, this also prevents any loaded program from being placed before this address. Possible usage of this setting includes preserving some room for future system services or inter process memory.

The base_progmaxsize settings is the primary means of assuring loaded processes will not overflow their allotted memory by selecting a value fitting to the largest needed memory range.

The MAXPROCS setting determines the size of the array which holds the proccess administration blocks. Effectively the maximum number of simultanious processes.

The true Maximum number of programs is determined by a simple caclution.

$$\frac{Memory_{available} - base\_progmaxsize}{base\_progmaxsize}$$

This enables the user to optimize for the per process memory needs and the desired amount of processes running at any time.

The PRINTCORE setting is used to pick the core used for blocking prints, as such it should not be used for performance intensive processes.

The MEMCORE setting is used to pick the core used for blocking memory accesses needed for atomic structure access, as such it should not be used for performance intensive processes.

[8]Some efforts of sabotage are predicted to be effective as detailed in Section 6.2.1

The NODE_BASELOCK setting is used to pick the core used for PID allocation, it should not be used for performance intensive processes.

Minimum page size is set by minpagebits, this is important should the target platform change.

Maximum page size is set by maxpagebits, this is important should the target platform change.

The identifier string for argument ELF section is determined by ROOM_ARGV, in the same maner ROOM_ENV does so for the environment ELF section.

2.3.2 Dynamic configuration parameters

The user can guide the loader by using a configuration file. The configuration file contains the following parameters:

- Filename of the ELF file (string)
- Arguments, can be an empty line (newline separated strings)
- Environment, can be an empty line (newline separated strings)
- Verbose, "true" or a numerical value"
- Exclusive, "true" or "false" (Optional, defaults to false)
- Core_start, numerical core number, (Optional)
- Core_size, numerical number of cores, (Optional, defaults to 1)

The filename setting is obligatory, the arguments and environment will need representation in the configuration although these may be represented by an empty line. The remainder of the settings are optional.

Settings Enumeration

Next to the configuration file, software can also offer flags as parameters to the loader API. These flags can be a combination of the following enumeration OR'ed together. The enumaration is defined in loader_api.h.

- e_noprogname, if true the argv[0] will not be passed based on the ELF file.
- e_timeit, if true print timing information on termination of a loaded process.
- e_exclusive, if true sl_exclusive is passed to the MGSim.

Our program manager offers an option to fully reserve the cores for the given program, blocking any core sharing. This setting is set via the e_exclusive flag.

Due to the serializing effects of this option and the absence of intelligent core selection this is disabled by default. By default the program will be started on the core invoking the loader call. A possible sideeffect of the e_exclusive option could be deadlock, if a program is started on a reserved core. The program would prevent any calls dependent on that core, including those it needs to terminate. A check could be implemented into the process manager prior to accepting this flag, in a performance critical section.

2.4 Implementation considerations and reflection

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

2.4.1 Placement of programs in the shared address space

Size

A program needs a location which can not be easily changed after the program has started. The run time of a program may be unbounded and as such a single program introduces fragmentation of 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[9].

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.

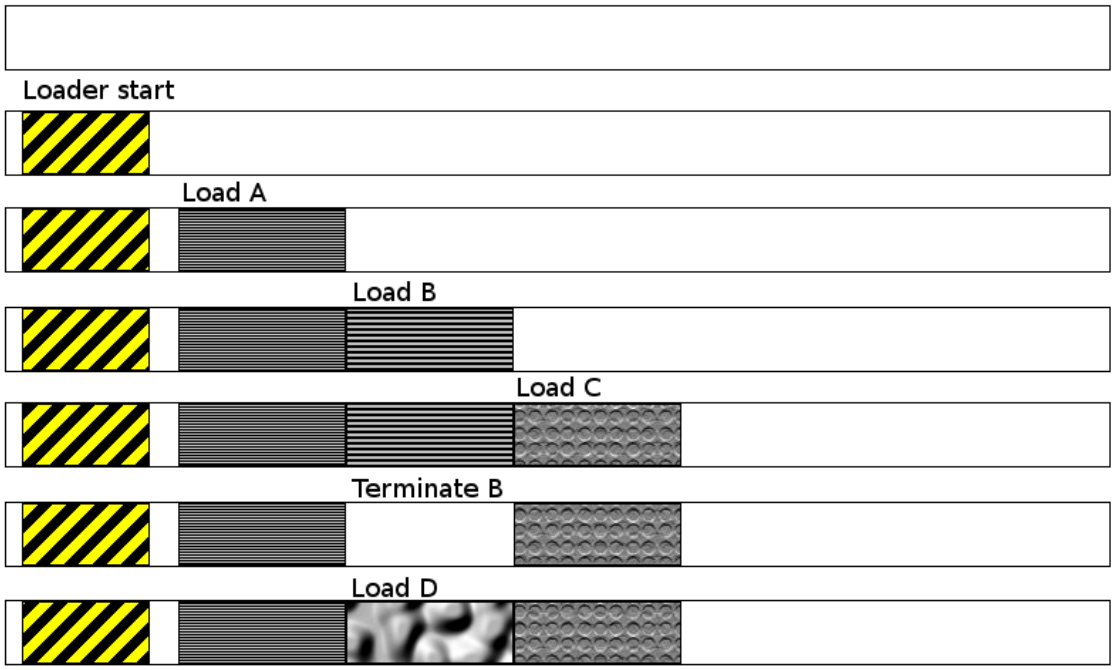
Solution

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 by a simple caclulation based on the Process Identifier.

Figure 2.2: Memory occupancy evolution



An indicattion of the resulting memory layout is shown in Figure 2.4.1, where the loader is shown to populate several regions of memory after each noted event.

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

By selecting a value for $Size_{maximumsubspace}$ the maximum amount of memory one loaded process can legally address is determined, in order to prevent overflow we have opted to use a default value of 2^{50} which offers each process more addressable memory than current platforms offer as randomly accessible memory. This value can be tweaked to suit any specific situation as defined in Section 2.3.1

2.4.2 Traceability of problems

As a system runs an increasingly large number of programs, problems tend to become more complex. We propose to use a memory allocation system which allows identification of the owning process by simple calcation, requiring an address and static configuration data. Our proposed method is computionally attractive as it does not require any other data or lookup. In order to trace errors we can determine ownership by calculating the Id of the memory based on the inverse of our location formula.

$$Id = rounddown((Address - Base_{Global})/Size_{maximumsubspace})$$

2.4.3 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. [4]

Summary

The loader loads initial programs via a configuration file, it allocates the needed memory and after programs terminate cleans everything up. All loaded programs are offered acces to the systems API which can be used to spawn other programs.

CHAPTER 3

Implementation

3.1 Assumptions and constraints

In the implementation process of the loader several assumptions had to be made.

- Availability of a working C compiler, slc
- Availability of a working linker, slcc
- 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 CSA group provides a C toolchain to program the Microgrid, fully compatible with the MGSim platform that we target in our work. This toolchain comprises of a C compiler which supports concurrency management extensions to C called SL. The compiler itself is called `‘slc‘` and uses GNU C as a back-end to produce code, as such slc is nearly fully compatible with C99 as supported by GNU C.

3.1.2 The linker

The process of grouping object code, produced by the compiler, together with libraries to form an autonomous executable file file falls under the responsibility of a linker program, commonly called `‘ld‘`.

Like gcc, the command slc can also be used to drive ld, ld accepts parameters to tune the linking process, such as whether to include relocation information in the final executable. We explore these in Section 3.1.3.

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 relocatable. The -shared flags indicated to the linker all data references 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.

3.2 API

The Application Programming Interface, an interface to be used to interact with and guide to loader. This is implemented in our loader via an interface akin to a UNIX syscall table, where all loaded program are offered a pointer from which at known offsets certain function pointers are stored. These functions are to be abstracted by the C standard library as they represent system calls directly into the loader which in this area could be seen as the operating system.

The loaded programs currently lack a full C library which offers functions such as fopen, fprintf and many others. The incompatibility with our loader stems from the preexisting C runtime used to link and locate the library 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:

Name First argument	Description Second argument	Return value Leftover arguments
spawn const char* ELFFileName	spawn a program enum settings	int 0 on success int argc, char **argv, char *env
print_string const char* PrintedString	prints in an orderly fashion (blocking) int WhichOutput	None
print_int int PrintedNumber	prints an integer in an orderly fashion (blocking) int WhichOutput	None
print_pointer void* PrintedValue	prints pointer in an orderly fashion (blocking) int WhichOutput	None
load_fromconf const char* ConfigFileName	loads a program from config file	int 0 on success
load_fromconf_fd int FileDescriptor	loads a program from config from an open file	int 0 on success
load_fromparam struct admin_s* PreparedStruct	loads from structure with parameters	int 0 on success
breakpoint int IdForPrinting	loader break point for program, prints and breaks const char* ForPrinting	enum WhoHandledIt

3.3 Platform dependency

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

- sl_create, for creating the program stack and core allocation, accepting paramters for core placement, exclusive core access and entrypoint
- sl_detach, letting the loader detach from a loaded program logically tied to sl_create.
- mgsim_control, for sending messages to the MMU concerning range allocation and deallocation, accepting parameters for address, size, permissions and PID
- mgsim_control, for sending a message to the simulator concerning a breakpoint

3.4 Configuration

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[1] 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 where 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
if Inspection of the header fails **then**
  Abort the loading
end if
Locate the program headers

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

Scan the program headers for the base address
$PID \leftarrow Available_{PID}$
$Base \leftarrow Align(basecalculation(PID))$
for all $Header \in Programheaders$ **do**
  $Dest \leftarrow Header.Location + Base$
  $Src \leftarrow Header.FileOffset$
  $Dest[0 : Header.InFileSize] = File[Src : Header.InFileSize]$
  $Dest[Header.InFileSize : Header.Memorysize] \leftarrow \{0, \ldots\}$
end for
$Relocations \leftarrow \{\}$
$SymbolTable \leftarrow 0$
Locate the section headers
for all $Header \in SectionHeaders$ **do**
  if $Header.type = Relocation$ **then**
    $Relocations \leftarrow Relocations, Header$
  end if
  if $Header.type = SymbolTable$ **then**
    $SymbolTable \leftarrow Header$
  end if
end for
for all $Symbol \in SymbolTable$ **do**
  if $Symbol.Name = ArgumentRoomName$ **then**
    $Argroom \leftarrow Symbol$
  end if
  if $Symbol.Name = EnvironmentRoomName$ **then**
    $Envroom \leftarrow Symbol$
  end if
end for
for all $Table \in Relocations$ **do**
  for all $Entry \in Table$ **do**
    $Loc \leftarrow Base + SymbolTable[Entry.Symbol].Offset$
    $Value \leftarrow SymbolTable[Entry.Symbol].Value$
    $Addend \leftarrow SymbolTable[Entry.Symbol].Addend$
    $*Loc \leftarrow Value + Addend + Base$
  end for
end for
if Argroom **then**
  Copy the Arguments into Argroom
end if
if Envroom **then**
  Copy the Environment into Envroom
end if
Call a new thread with the Entrypoint, Arguments and Environment

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,
$Key \leftarrow$ ""
$Value \leftarrow$ ""
$Buffer \leftarrow$ ""
while $X \leftarrow NextCharacter$ **do**
  if $X = '='$ **then**
    $Key \leftarrow Buffer$

```
      if X = NEWLINE then
        if Key = "" then
          Break While
        end if
        Value ← Buffer
        Call ParseSetting(Key, Value)
      else
        Buffer ← Buffer, X
      end if
    end if
  end while
```

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
for all X ∈ ToRead do
  if X = MEWLINE then
    if Argv[Argc][0] = NULLBYTE then
      Goto Done
    end if
    Argc ← Argc + 1
  end if
  Argv[Argc] ← Argv[Argc], X
end for
```

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, to tell the compiler to generate position independent code.
- -fPIC, to tell the compiler to generate position independent code which could be needed for compilation to SPARC machines[2].
- -shared
- crt_fun.c
- -nostdlib

These flags tell slc [3]to compile position independent code, to include data relocation information.

By default slc links programs with a simple bootloader and operating system, which assume full control of the chip to that program. The flag -nostdlib prevents this automatic bundling, and allows us to link our own initialization code. Which simply calls the 'main' program of the 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

[2]it concerns GOT maximum size

[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.
```

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. 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.

The loader lacks dynamic library support, programs 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.

Our loader does 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 further programs recursively.

Another function is the powerfull load_fromparam which allows loading with fully customized settings without the need for creating writing to a configuration file.

CHAPTER 4

Progress report

As our research progressed we ran into some obstacles.

4.1 Milestones

In order to guard completion of our research we selected several milestones, each of which designed to be an improvement over those before.

4.1.1 Planned milestones

A loader for single program, this milestone was picked due to the importance of the loader component of our system. Without being able to load a simple program the rest of our research would be impossible. This has since completion formed the basis of the ELF loader of our system.

A loader for multiple programs, as our research progressed we introduced those components needed for the loading of more than a single program. These components though not of much use on their own cooperate with our ELF loader to allow the loading of programs which can be deterimed at runtime. This collection of components is the core of our research.

A loader with in program spawn function, as the system progressed the need for dynamic loading increased. In order to allow loaded software to load other programs we introduced an API which can be utilized by software such as shells.

A loader with Input and Output redirection, as the number of loaded processes increases the need for regulated input and output arrises. Processes are offered functionality to print to both the standard output and standard error streams of the system. This demonstrates the posibility for the system to offer true IO via out API.

4.1.2 Unexpected roadblocks

Missing functions, some of the functionallity required for the allocation of memory were not readilly available. The interface we used did not offer dynamic allocation of Executable memory, we have had to temporarilly moddify the protections settings of the simulator in order to complete our research. Our solution effectively makes all allocated memory executable.

Libc conflicts, loading an existing libc leads to crashes. The toolchain we used during our research has been adopted to initialize the full system on program startup, this is done by including a simple bootloader in compiled programs. This bootloader had to be disabled by passing a flag to the compiler and linker as noted in Section 3.5.3.

Runtime cleanup, programs were not being cleaned due to thread termination. Our initial replacement bootloader for the loaded programs included both some initialization and cleanup code. An unwanted sideeffect of the cleanup code was the termination of the threads the program was running. The termination effectively meant that any program which was loaded could not be cleaned as the process manager was never signalled that a program was finished. The solution

to this problem was replacement of the overzealous bootloader by one that simply called the main function and returned its return value.

4.1.3 Reached milestones

As our research has progressed so have the intended milestones, resulting in a functional system capable of running programs in parallel and reusing any shared resource.

As we have reached our intended milestones we need to acknowledge some other achievements.

The loading of a relocatable program, which requires specific data relocation.

We have implemented loading based on a simple configuration file and in program API calls. This enables users to load programs either by hand or automaticly. These methods of loading also let the user start their new program on a specific core.

4.2 Future research

4.3 Security

As the memory manager is focused on sharing an address space between programs some assumptions where made as seen in Section 6.2.1.The platform could be extended with 'capabilities' to prevent rogue threads accessing information the may not, this would however need to be supported by hardware much like a TLB.

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 Benchmarking results

During the benchmarking of our system we have run several programs. Each run of these programs started a number of programs with specific behaviour and timing information enabled. The bulk of these programs was started by our test program called `sparmy`. The resulting output and error streams were stored to logfiles for offline processing. The timing data is printed prior to program termination in a format designed for simple parsing by a python script which we tasked with visualization. The specific format is `<Clocks>%d,%d,%d,%lu,%lu,%lu,%lu</Clocks>` where the following items apear in this order:

- PID, the process identifier
- Core_start, the core the program was started on
- Core_size, the number of allocated cores for this program
- CreateTick, the absolute value of Clock at creation, used as tick zero for the elapsed ticks
- TicksToDetach, the number of elapsed ticks to the actual program entry
- TicksToEnd, the number of ticks elapsed to the programs return
- TicksToCleaned, the number of ticks elapsed to the program being cleaned up

Figure 1 was gatheredfrom running our test program which called 256 programs. The used programs were compiled from C sourcecode which on the call to main, returns 0 right away. This gives us an indication of performance of the loader, process manager, memory manager and printing of timing data.

As it can be observed initial performance shows some irregularities which weaken as more than 64 programs have been loaded. The initial peaks in performance can be observed more closely in Figure 2, where it can be seen that after the highest core used in our test, which is core 127, the performance stabalizes. We speculate this bahaviour is due the needed initialization of cores and L2 caches. The latter due the recurring spike each fourth core.

In Figure 4 the creation of the tested programs shows linear performance. The program used for this measurement was designed to have a stable runtime which exceeds the overhead observed for empty programs.

CHAPTER 6

Conclusions

6.1 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.

6.2 Limitations and future work

6.2.1 Permissions

In order to explore possible problems extra test programs for corner cases were constructed. These have been used during testing to improve the loader. There is however another class of programs, malicious and erroneous programs, which attempt to access memory which was allocated for another loaded program or even the loader itself. We did not consider this class as our platform did not support access control yet.

A means of protecting loaded programs from each other is documented in [5]. Their proposed protection system would enable fine grain access control and secure the loader and programs from ill written programs if not malicious programs.

6.2.2 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.

Programs could all use the same copy of program code, where a loader would fill in all unresolved symbols by locating the symbols from single copy of the library into each program being loaded.

6.3 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.

In other words, the implementation of a loader program is a foundational stepping stone which bootstraps the implementation of a fully fledged Operating System on this platform.

6.4 Final conclusion

Our system has shown that a collection of seperatly compiled programs can run in parallel harmony, the loaded programs themselve can request the loader to load more programs and these will continue their execution even after their invoking process has finished. The process identifiers used to mark processes can be efficiently recycled, allocated memory is cleaned up after a process has terminated as to ensure the loader can remain active after initial processes have terminated. We predict the system can load as many processes as the available memory allows, the upper limit for the amount of processes can be influenced by adjusting the amount of reserved per process memory range and setting the apropriate setting in MAXPROCS which is found in elf.c.

Bibliography

[1] Jeff Chase, Miche Baker-Harvey, Hank Levy, and Ed Lazowska. Opal: A single address space system for 64-bit architectures. *SIGOPS Oper. Syst. Rev.*, 26:9–, April 1992.

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

[3] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, 1998.

[4] 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.

[5] 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.

.1 Terms

- OS, Operating System
- MMU, Memory Managment Unit
- TLB, Translation Lookaside Buffer
- CLB, Capability Lookaside Buffer

.2 Benchmarking figures

Figure 1: Program spawning empty programs

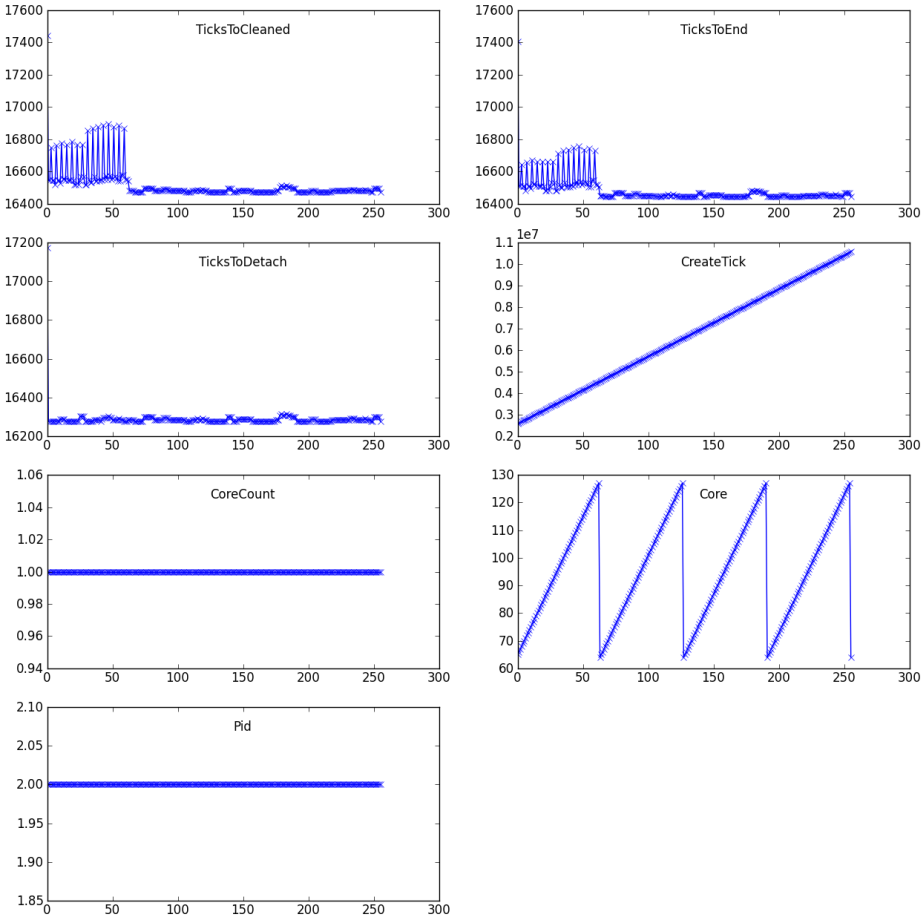


Figure 2: Program spawning empty programs, closeup

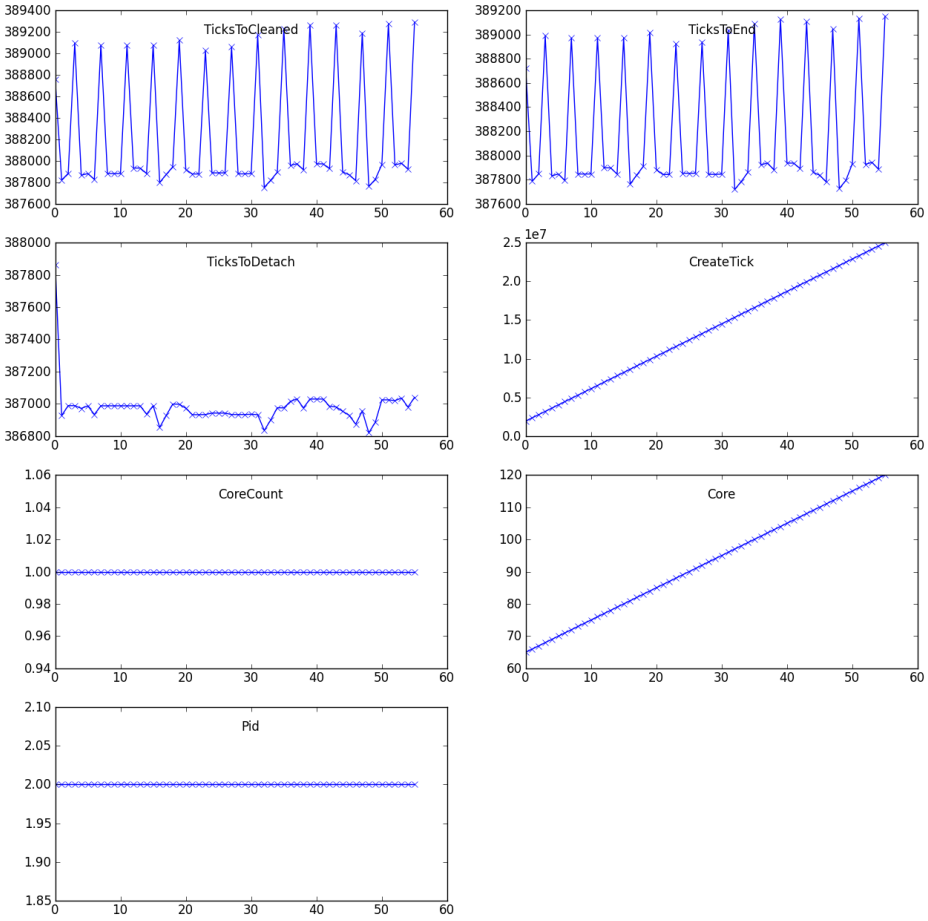


Figure 3: Program spawning simple printers

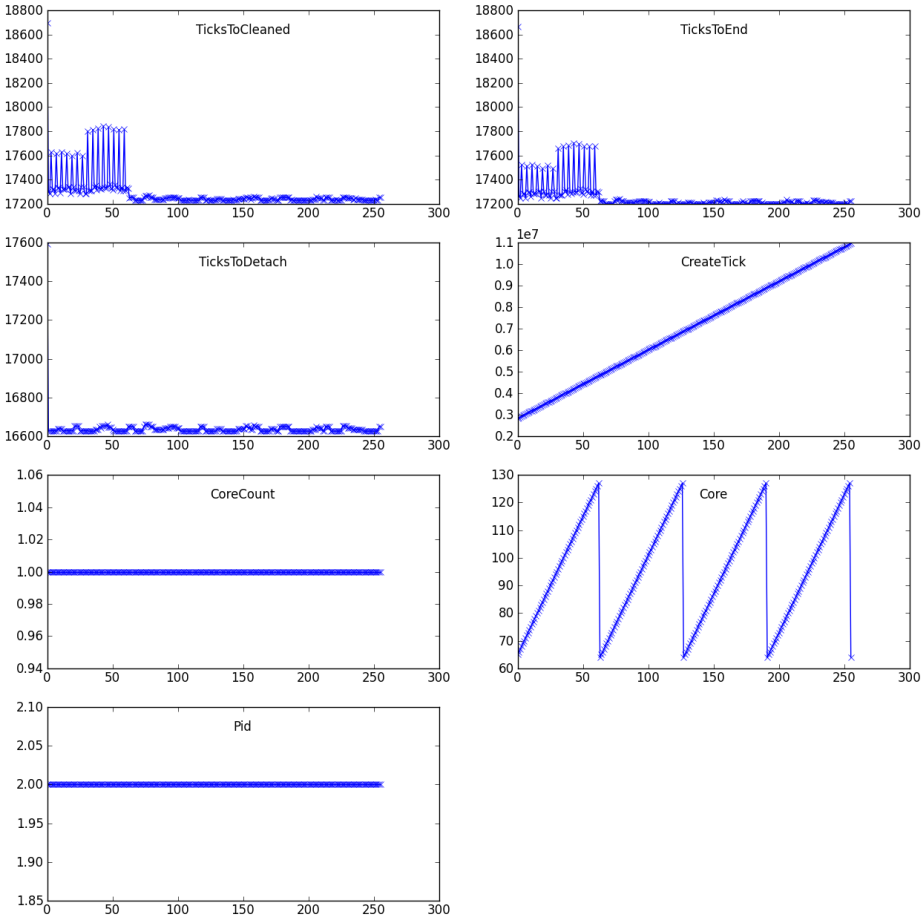
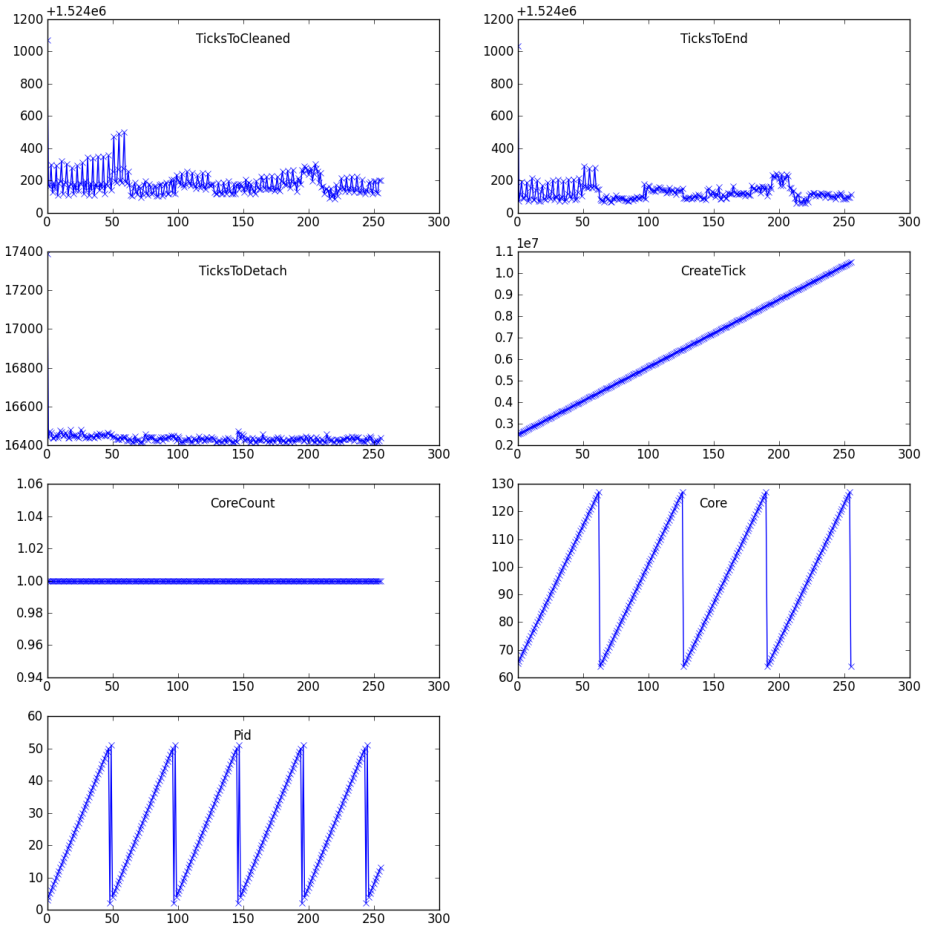


Figure 4: Program spawning counters



APPENDIX A

Problems and in depth solutions

A.1 Bugs

A.1.1 Implementation details

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
```