Emacs Portable Dumper

Daniel Colascione

March 14 2018

About me

- Day job: Android performance team
- For me: happiness slopes away from ring 0
- Emacs development: both tool refinement and hobby
- Got into developing the core as part of customizing environment

What is Emacs?

- Text editor
- Mail reader
- Document preparation system
- Tetris platform
- Text adventure
- Floor wax

What is Emacs, really?

- Runtime environment
  - Garbage collector
  - Interpreter
  - Compiler
  - Program loader
- Lisp system
  - Intimate relationship between development, use
  - Save and restore whole system state
  - Closest modern analog might be IPython notebook

Build and run overview

- Emacs dumps itself during build process
    1. Build system makes proto-emacs called `temacs`
    2. `temacs` loads `loadup.el`, which loads Emacs core
    3. Create `emacs` executable from resulting process state
- On `emacs` start, it's as if loadup had already happened
    - Almost literally true
        - Can't store open files
        - Can't restore open windows

Why dump? Performance!

- From scratch

```
$ time ./temacs -batch -Q  --eval '(kill-emacs)' \
     2>/dev/null
real    0m4.946s
```

- Dumped

```
~/edev/trunk/src
$ time ./emacs -batch -Q  --eval '(kill-emacs)'
real    0m0.036s
```

Why care about performance?

- Isn't slow startup acceptable?
    - No: Emacs is often EDITOR: needs acceptable latency for light cases
    - Startup snappiness affects perception of general performance
    - Previous slide is just core: packages can take much longer

Why care about performance?

- Can't we use the Emacs daemon?
  - Fine for some use cases: but requires setup
  - Shared environment not necessarily desirable
  - Persistent bloat: what if all programs did this?

Modernizing Emacs

- `Unexec` is traditional dump implementation
  - Clever, but showing its age: 36 years old!
  - Dubious long-term maintainability
- Replacement: `pdumper`
  - Goal: get rid of old `unexec` code
  - Requirements
    - no loss in performance
    - no loss in capability
    - reliance on normal, supported facilities that will keep working
  - Goals achieved!
    - Did most work in 2016
    - Finished a few months ago
    - Waiting for merge into mainline

Dumping in Lisp systems

- Emacs conceived as Lisp system
- Lisp system tradition: dump and restore
    - Capability dates back to 1960s
    - Even modern Lisp systems like Allegro and SBCL have dumpers
    - Emacs came from AI, lisp machine environment
- Lisp systems had deep introspection support
    - Like Emacs, but for the whole OS, kernel and all
    - Dumping just an application of introspection

But unix was `void*` and without form

- GNU Emacs needed to run on Unix
- **PROBLEM!** Unix had zero introspection!
  - Bare-bones process abstraction
  - Just a bunch of bytes
  - No global dump and restore support
    - Core dumps don't count
  - Lisp could run in a started process: but no startup help
    - Just imagine how long loadup took in the 80s!

Unexec to the rescue

- Unexec is a clever hack for implementing Lisp-style dumping on Unix using a bare minimum OS help
- Elegant and simple: takes advantage of details of existing executable loader and file format
- Fortunate Emacs had it: Unix won utterly
  - Pre-Unix OSes are like Precambrian biota
  - Weird, wonderful, and forgotten

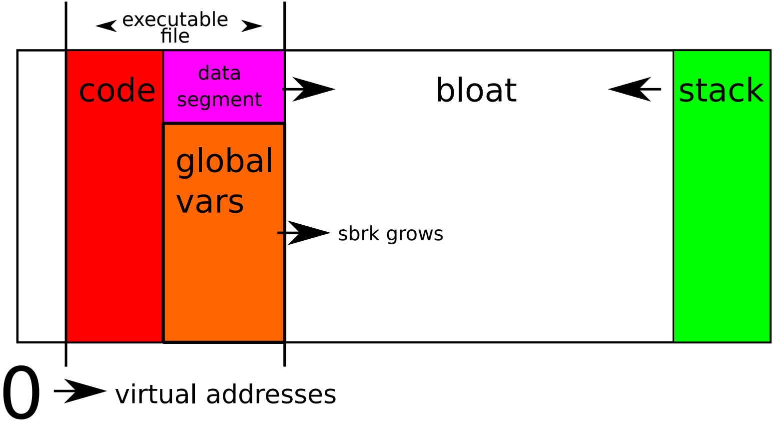
Program loading

- First need to understand how programs run
    1. Kernel creates blank address space
    2. Kernel causes executable file to appear at known address in the new address space
    3. Kernel initializes a task structure
        - Registers set to known values
        - Program counter begins at well-known address inside the program
    4. Kernel context-switches to new task and begins executing
- Same basic model used today

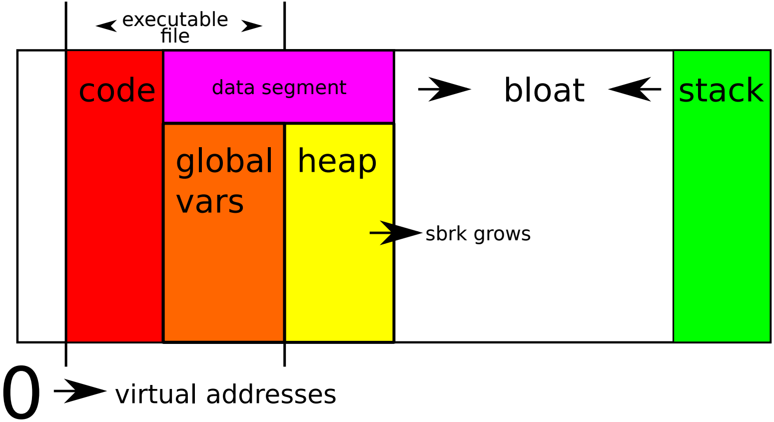
Address space structure

- Executable code (aka "text") appears at address T
- Data (variables, bss, etc.) appears at T+size(text)
  - Values come directly from executable file!
- The stack starts on other end of the address space
- Dynamic memory allocation is accomplished by growing the data region
  - Data region grown as needed using `sbrk`
  - Malloc implementation carves out chunks of new memory

Normal address space layout: brand new process



Normal address space layout: active process



Unexec operation

- `temacs` starts and runs normally
  - loads `loadup.el` and does a bunch of work
- After this process completes, the process has
  - changed global variables in bytes mapped to `temacs` executable
  - expanded its data segment to accommodate dynamic memory allocation (see previous diagram)

Unexec's central trick

- To make a dumped `emacs`, unexec
  1. Copies `temacs` to `emacs`
  2. Modifies `emacs` so its on-disk data segment size is the size of the current **in memory** data segment size of the `temacs` `process`
  3. Copies the current `temacs` data segment to the new enlarged data segment in the `temacs` executable
- This way, the new executable "freezes" the result of whatever it is that `temacs did`
  1. Whatever `temacs` did, it's reflected in the heap or in changes to global variables

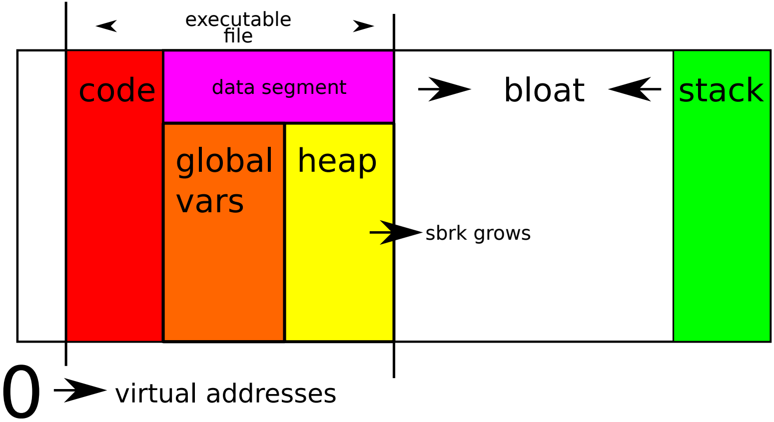
Running a dumped Emacs

- When the new `emacs` process executes, the kernel goes through its normal logic
  - Maps data segment into memory...
  - ...automatically mapping the initialized heap!
  - The last value of any global variable that `temacs` set appears to be that variable's **initial** value in `emacs`!
- Heap grows normally as `emacs` runs.
- The "restore" is just the normal operation of normal executable loading.

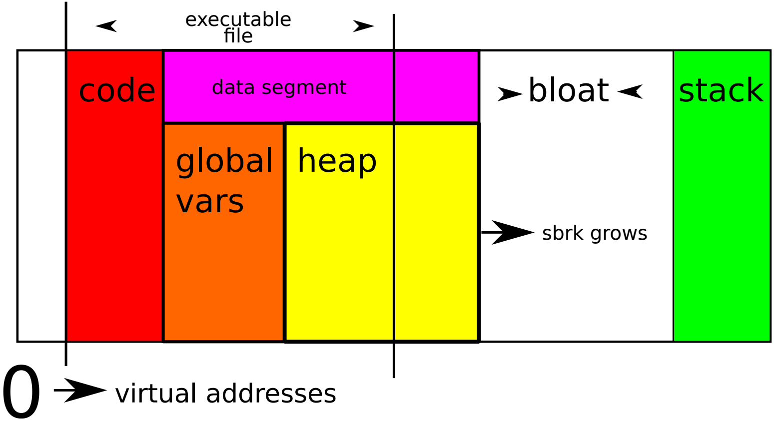
Why does it work?

- All `temacs` pointers still valid in `emacs`!
  - No pointers to old stack
  - Pointer to text? Same spot in memory
  - Pointer to globals? Same spot in memory
  - Pointer to the heap? *Same spot in memory*
- `main` function in `emacs` can detect it's running in a dumped emacs: `initialized` global != 0
  - Re-open file descriptors
  - Connect to window system
  - Perform other necessary adjustments

Unexec address space: just started



Unexec address space: active process



Why is unexec a good hack?

- Minimal
  - Complexity is all on the dumping side
  - Initial implementation from 1982 was only about 300 LOC
- Theoretically optimal speed
- Surprisingly portable: same basic approach works on everything from Windows to HP-UX
- Surprisingly long-lived: at least 36 years

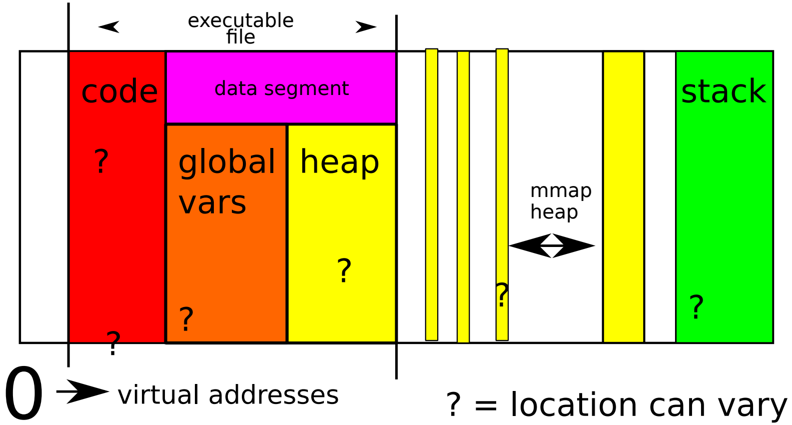
Unexec must go

- Complexity: now almost 5,000 LOC
- Obscure
- Most importantly, insecure

Unexec complexity

- Hairy platform-specific code to munge executables
- Many different sections and segments compared to a.out's two
    - Random whitelists of dumped section names
    - What if we miss one? Random crashes
- Dynamic linker assumes it sees straight-from-compiler code
    - Need to "undo" relocations so re-doing them is a no-op
    - Depends on platform

Modern address space



Unexec obscurity

- Re-dumping code bitrotted years ago
- Unexec relies on internal malloc hooks
  - malloc state needs separate dump, restore
  - glibc trying to remove API
- Incompatible with modern malloc implementation
  - Either `temacs` needs to force malloc to be `sbrk`-only malloc, or...
  - `temacs` needs to use separate, internal malloc implementation...
  - ...and switch dynamically. Yuck.
- Platforms not designed for unexec, so weird breakages

Who wants to spend time working around BSS gaps?

```
  /* Warn if the gap between BSS end and heap start is larger than this.  */
# define MAX_HEAP_BSS_DIFF (1024*1024)

  if (heap_bss_diff > MAX_HEAP_BSS_DIFF)
    {
      fprintf (stderr, "**************************************************\n");
      fprintf (stderr, "Warning: Your system has a gap between BSS and the\n");
      fprintf (stderr, "heap (%"pMu" bytes).  This usually means that exec-shield\n",
               heap_bss_diff);
      fprintf (stderr, "or something similar is in effect.  The dump may\n");
      fprintf (stderr, "fail because of this.  See the section about\n");
      fprintf (stderr, "exec-shield in etc/PROBLEMS for more information.\n");
      fprintf (stderr, "**************************************************\n");
    }
```

Security disaster: unexec ^ ASLR

- Unexec requires run-to-run memory layout consistency
  - Otherwise, dumped pointers are invalid
- Address Space Layout Randomization requires address space layout be *different* every time
  - Otherwise, attackers can exploit memory corruption bugs
- **Unfixable**

A different kind of dumper

- Want to preserve dump model while ditching unexec
- Fundamental problem is that pointers need to point different places on each load
- We'll just teach Emacs to relocate its own pointers
  - Dump objects, not "the heap"
  - Record all pointer locations
  - Munge every pointer on load
- Should work on any system with any file format
  - Need to restrict ourselves to "happy path" of loading
    - No weird sections
    - No weird permissions
    - No weird malloc modes

New dump process

- After `temacs` loadup
  1. Walk the Emacs heap (just like during GC)
  2. Dump raw object contents, struct by struct; remember where we dumped each
  3. Remember each pointer and where it points
     - If into Emacs, write the offset into Emacs
     - If into the dump, dump offset of pointed-to object
  4. Write the values of all global variables and their offsets relative to the Emacs executable
  5. Write the pointer list to the dump

New load process

- On `emacs` startup
  1. Very early in `main`, load or map the dump into memory
  2. Walk the list of pointers in the dump and adjust each one
     - If point into Emacs, adjust by current offset of Emacs executable
     - If point into dump, adjust by actual dump load location
  3. Set all global values to the values stored in the dump
  4. Allow initialization to proceed

Like an executable if you squint

Dump section: header

Header Metadata about dump

- Magic number
- Emacs fingerprint
- Table offsets

Dump section: hot

Hot Primary heap contents

- Objects in this section need relocation
- Relocations apply here
- Mark bit array covers only this section

Dump section: discardable

Discardable Thrown away after Emacs starts

- Shadow objects we copy into Emacs executable (like symbols)
- Relocations apply here too

Dump section: cold

Cold Things that don't need relocation and that we can easily share between Emacs instances

- Objects with no internal lisp pointers
  - Floats
  - Bool vectors
- Pure data
  - String data
  - Buffer contents
- Relocation tables

Dump Relocations (1/2)

```
enum dump_reloc_type
  {
    /* dump_ptr = dump_ptr + emacs_basis() */
    RELOC_DUMP_TO_EMACS_PTR_RAW,
    /* dump_ptr = dump_ptr + dump_base */
    RELOC_DUMP_TO_DUMP_PTR_RAW,
    /* dump_lv = make_lisp_ptr (
         dump_lv + dump_base,
         type - RELOC_DUMP_TO_DUMP_LV)
       (Special case for symbols: make_lisp_symbol)
       Must be second-last.  */
    RELOC_DUMP_TO_DUMP_LV,
    /* dump_lv = make_lisp_ptr (
         dump_lv + emacs_basis(),
         type - RELOC_DUMP_TO_DUMP_LV)
       (Special case for symbols: make_lisp_symbol.)
       Must be last.  */
    RELOC_DUMP_TO_EMACS_LV = RELOC_DUMP_TO_DUMP_LV + 8,
  };
```

Dump Relocations (2/2)

```
#define DUMP_RELOC_TYPE_BITS 4
#define DUMP_RELOC_ALIGNMENT_BITS 2
#define DUMP_RELOC_OFFSET_BITS                          \\
  (sizeof (dump_off) * CHAR_BIT - DUMP_RELOC_TYPE_BITS)

struct dump_reloc
{
  uint32_t raw_offset : DUMP_RELOC_OFFSET_BITS;
  ENUM_BF (dump_reloc_type) type : DUMP_RELOC_TYPE_BITS;
};
```

Lisp API

`dump-emacs-portable` Dumps current Emacs image to file

`pdumper-stats` Returns list describing dump metadata, load time, etc.

dump-emacs-portable operation

- Chew through a big queue of objects
- Queue initialized with GC roots
- Heuristic tries to keep related objects together
  - "Rubber band" weight attached to each link
  - Pulls objects from queue into dump
- Similar to GC, but actually very different
  - We can allocate memory during dump
  - Unlike GC, we care about all of the object, not just lisp fields

C API

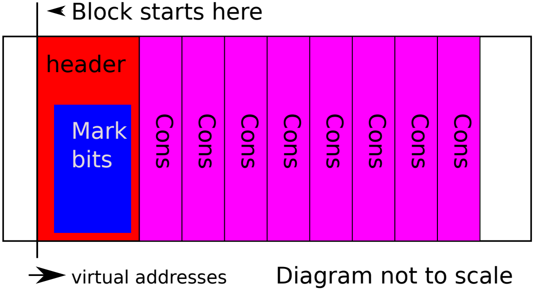
- Global variables: most Just Work
  - Automatically record each GC root
  - Automatically record anything DEFVARed
  - Need to call into pdumper in special cases, e.g., remember a scalar
- Post-dump callback
  - Call function using `pdumper_do_now_and_after_load` from `syms_of_`
  - In non-pdumper build, calls function right away
  - In pdumper build, given function automatically called after dump restore

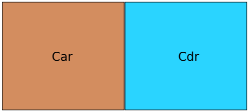
Early failures

- I implemented this basic dumping strategy
- Emacs crashed and burned right away
  - Refactor and rearrange early init code
  - Use different GC strategy for pdumped objects
  - Separate list of object-start relocations for conservative GC
  - Special treatment of hash tables

Allocation in normal execution

- Emacs allocates popular object types in blocks
- GC zeroes low bits to find header

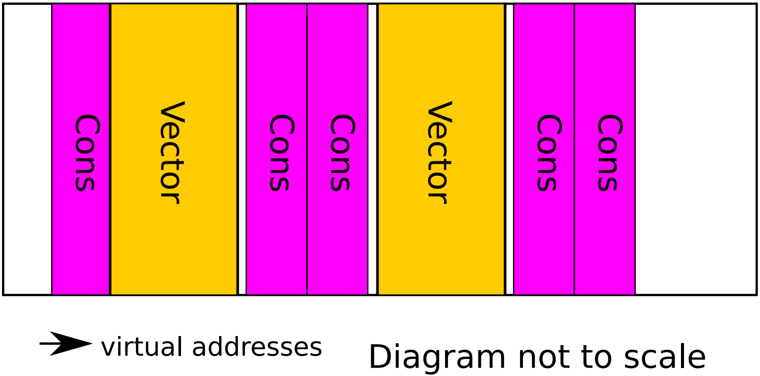




Cons Cell

Object layout in pdumper

- Pdumper dumps object-by-object
- No header: objects of different types can interleave



Making garbage collection work with pdumper

- GC crashes when trying to mark pdumper objects
  - Reads garbage as header
  - No place to read or write mark bit
- Solution: better than original book-keeping approach!
  - Keep one big bit-array of mark bits for whole pdumper
  - Simple range check lets GC distinguish dumped objects from heap objects
  - Better than individual mark bits: easier to clear; return memory to OS
  - **No copy-on faults just for GC** (better than unexec)

Conservative GC overview

- Emacs used to use precise stack marking via complex CPP macros
- Got rid of them: uses conservative scanning instead
- Treats all words on stack as potential pointers into the heap
- Detect valid objects by keeping a big red-black tree of known memory regions
- Pdumper has no such memory region tracking: no blocks, no metadata

Pdumper introspection

- To cooperate with conservative GC, need to be able to find object-start
- Turns out the relocation table is exactly the right data structure
  - Fake relocatons that describe object starts and types
  - Sorted for fast lookup during stack scanning

Annoying introspection bug

- Bug! Early versions validated object start, but forgot to check object tag bits
- Took a few days to find: reproed only occasionally
- Would accidentally treat buffer as float or something
- Solution is to check both object address and type when considering a candidate `Lisp_Object` from stack

Hash table bug

- Some hash tables would retrieve wrong hashed objects
- Some objects are identity-hashed: hash code is memory location
- Not feasible to use Java-style identityHashFunction across dump
- But we *can* rehash hash tables
- Negative size: we must rehash

RR is awesome

- Aside: RR tool is awesome
- From Mozilla: reverse debugging
- Record and replay execution
- Makes it easy to answer question "who produced this bad value?"
- Probably halved pdumper development time

Demand paging?

- Dump relocated all at once on startup
- What if we could relocate each page as needed? Start in microseconds!
- Can hook SIGSEGV and run code just before we read a dump page

Demand paging? Not worth it

- I wasted a lot of time implementing demand paging. It's useless!
- Why?
  - We GC a ton
  - GC doesn't COW, but it does have to load pages read-only
  - Relocated pages are then COWed
  - First GC touches 90% of dump anyway
  - Might as well get startup over with: only takes a few dozen milliseconds
- No clear way to traverse GC graph in much less space than heap itself

Fast non-PIC startup?

- Portable dump works great for randomized address space
- Works fine for old-fashioned non-randomized address space too, but wasteful
    - Unnecessarily relocates: relocated data known ahead of time
    - Unnecessarily takes COW faults during relocation
- Idea: if we know memory layout in advance, just write correct values directly to dump
- Save 6MB or so plus a few dozen milliseconds on startup

Fast non-PIC startup? Not worth it

- Turns out non-PIC mode isn't worth it
- Regular code is surprisingly fast
- Hard to justify PIC mode complexity
- Hard to guarantee fixed address even without PIC
- Can still implement non-PIC mode if needed, but probably won't be

Portable dump inside Emacs executable?

- Pdump dump is a separate file
- Separate file is annoying: can become mismatched
- Every known OS supports appending a blob to the end of an executable
- On startup, Emacs would open itself, seek to end, read header, seek to real header, load

Portable dump inside Emacs executable? Not worth it

- Turns out, `strip(1)` removes the dump from the file
- Appending dump would disturb digital signature: we don't sign now, but might one day

Questions

Questions