PLAN Virtual Machine for AMD64 Implementation Manual

Benjamin Summers and Elliot Glaysher

February 19, 2026

Contents

Abstract

This document explains the implementation of the AMD64 version of the PLAN Virtual Machine. This literate document contains the assembly code with documentation and commentary, while also presenting higher level whys. A reader should understand the memory layout and registers conventions used by the native runtime system.

1 Formatting Conventions

1.1 Assembly Code

Assembly code uses tabs for indentation, and tab between the instruction and the arguments. In the rare cases where the instruction is too fat for this (e.g. `lock cmpxchg`), just do whatever you feel like.

Code is written in two columns with pseudocode on the left, aligned to column 45 using spaces. The pseudocode is immensely helpful as a quick reference to describe what is happening on each line, and incredibly helpful when stepping through code in a debugger.

For aesthetic reasons, every line of assembly should have a # at column 45.

More complex explanations should be written in prose, using LaTeX instead of comments.

```
label:                                      # label:
        mov     rax, rdi                    #   res = x
        add     rax, rsi                    #   res += y
        ret                                 #   return res
```

1.2 C Code

C code is used for tests and for subsystems which are still being prototyped. Writing assembly is relatively painless, but refactoring it is incredibly time consuming.

C code should be indented using tabs, and aligned using spaces (but avoid aligning things).

2 Register Conventions

2.1 System V Amd64 Register Conventions

Just for reference, here are the normal conventions for this platform:

```
ARGS: rdi rsi rdx rcx r8 r9
RETURN: rax, rdx
TEMPORARY: rax, r10, r11
PRESERVED: rbx rbp r12-r15
SPECIAL: rsp rbp rip
```

These conventions are not respected, but are still used as the default conventions on which the actual conventions are based.

2.2 Global Register Allocations

Except in specific contexts which need access to a lot of different registers and don't need to allocate, these registers are always pinned to global variables:

```
r12 -- scratch buffer, ignored by GC.
r13 -- pointer to the first word immediately after the heap.
r14 -- pointer to the next free word on the heap.
r15 -- pointer to topmost item of the shadow stack.
```

While it is still possible to use r12 in contexts which require a lot of registers, a lot of macros clobber r12, so you must be careful.

2.3 Calling Pins and Laws

Pins and Laws share a single calling convention, which is similar to the normal amd64 but extended with self-reference, and with no callee-saved registers.

```
SELF: rax -- self reference (pin or law)
ARGS: rdi rsi rdx rcx r8 r9
OUT: rax
```

TODO: It may be worthwhile to extend the set of arguments used for registers here, since we clobber all registers anyways.

2.4 Calling Thunks

Thunks are evaluated by calling (or jumping to) the code pointer stored in the first slot of a thunk. Thunks contain all of the information they need in order to be evaluated, and they clobber all registers. The calling convention is just:

```
SELF: rax
OUT: rax
```

2.5 Calling Asm From C

We use some C code for debugging tools, and for some things that we are still prototyping. But C code can leave registers in invalid states, so we need to use adapters in order to correctly call into the runtime system from C.

```
.global zzcal2
zzcal2: jmp zcall2

.global zcall0, zcall1, zcall2, zcall3, zcall4, zcall5

zcall0: xor     rsi, rsi
zcall1: xor     rdx, rdx
zcall2: xor     rcx, rcx
zcall3: xor     r8d, r8d
zcall4: xor     r9d, r9d
zcall5: sub     rsp, 32
        mov     [rsp], r12                  # save r12, rbx, rbp
        mov     [rsp+8], rbx
        mov     [rsp+16], rbp
        mov     rax, rdi                    # self=arg1 (function pointer)
        mov     rdi, rsi                    # arg1=arg2
        mov     rsi, rdx                    # arg2=arg3
        mov     rdx, rcx                    # arg3=arg4
        mov     rcx, r8                     # arg4=arg5
        mov     r8, r9                      # arg5=arg6
        xor     r9d, r9d                    # clear r9
        xor     r10d, r10d                  # clear r10
        xor     r11d, r11d                  # clear r11
        xor     r12d, r12d                  # clear r12
        xor     ebx, ebx                    # clear rbx
        xor     ebp, ebp                    # clear rbp
        call    rax                         # Call into asm routine
        mov     r12, [rsp]
        mov     rbx, [rsp+8]
        mov     rbp, [rsp+16]               # restore r12, rbx, rbp
        add     rsp, 32
        ret
```

3 Convenience Macros

3.1 pdrop

```
.macro pdrop n
        .set bytes, 8 * \n
        add     r15, bytes
.endm
```

Drops a certain number of items from the shadow stack.

3.2 ppush

Pushes zero or more registers to the shadow stack.

```
.macro ppush regs:vararg
    .ifnc "\regs", ""

        .set n\@, 0
        .irp r, \regs
        .set n\@, n\@ + 1
        .endr

        .set total, 8 * n\@

        sub     r15, total

        .set offset\@, 0
        .irp r, \regs
                mov     qword ptr [r15 + offset\@], \r
                .set offset\@, offset\@ + 8
        .endr
    .endif
.endm
```

3.3 ppop

Restores a sequence of registers from the shadow stack. Note that the order corresponds with the order using for ppush. If you ppush a, b, you also ppop a, b.

```
.macro ppop regs:vararg
    .ifnc "\regs", ""

        .set n\@, 0
        .irp r, \regs
        .set n\@, n\@ + 1
        .endr

        .set total, 8 * n\@

        .set offset\@, 0
        .irp r, \regs
                mov     \r, qword ptr [r15 + offset\@]
                .set offset\@, offset\@ + 8
        .endr

        add     r15, total

    .endif
.endm
```

3.4 Block Saves

These are only used in the division logic, in order to push multiple values to the C stack. This should probably eventually be replaced with a generic macro like ppush and ppop.

```
.macro pop_r8_to_rdi
        pop     r8
        pop     rcx
        pop     rdx
        pop     rsi
        pop     rdi
.endm
```

```
.macro push_rdi_to_r8
        push    rdi
        push    rsi
        push    rdx
        push    rcx
        push    r8
.endm
```

3.5 sub0

sub0 is unsigned subtraction with a floor at 0. If the subtraction would underflow, the result is zero.

```
.macro sub0 a, b
        cmp     \a, \b
        cmovb   \a, \b
        sub     \a, \b
.endm
```

3.6 cap

cap(r,max) just computes r=min(r,max) in order to keep r within a certain bounds. Both inputs must be in registers.

```
.macro cap r, max
        cmp     \r, \max
        cmova   \r, \max
.endm
```

3.7 jzero

Jump to a label if a register is equal to zero.

```
.macro jzero reg, label
        test    \reg, \reg
        jz      \label
.endm
```

3.8 btw

```
.macro btw r1                                   # bits_to_words(x) = (x+63)>>6
        add     \r1, 63
        shr     \r1, 6
.endm
```

3.9 swapq

Using two temporary registers r1, r2, swap the values behind two locations in memory.

```
.macro swapq r1, r2, m1, m2
        mov     \r1, \m1
        mov     \r2, \m2
        mov     \m1, \r2
        mov     \m2, \r1
.endm
```

3.10 swapstk

Using two temporary registers r1, r2, swap the two values on the stack.

```
.macro swapstk r1, r2
        swapq   \r1, \r2, [r15], [r15+8]
.endm
```

4 Pointer Tagging

Numbers that fit within 63-bits are stored unboxed, directly in the value. A high bit being set indicates that the value is a heap-reference.

Heap-references are represented as normal pointer but with tagging data stored in the high 16 bits. The following conventions are used:

```
u63 0nnnnnnnnnnnnnnn = 0x0000..0x7fff (n=nat data)
NAT 1000000000000000 = 0x8000
PIN 11000100000000cm = 0xC40x (c=hascrc32, m=ismegapin)
LAW 1100100000000000 = 0xC800
CLZ 1101000ttttssss  = 0xD0xx (t=tag, s=size)
THK 1110000000000000 = 0xE000
```

the m flag indicates that a pin is a megapin, which is a pinned pin. Megapins cache information about all of their subpins.

the c flag indicates that the pin has been hashed using crc32 (which is directly supported by the CPU). Pins that have this information are bigger, having the hash data append to the end of the structure.

If the closure tag information is 0b1111, that indicates that the tag is too big and must be loaded from the closure itself.

If the closure size is 0b0000, that indicates that the size is too big and must be loaded from the closure itself. 0 is chosen because 0 is not a valid closure size.

4.1 Essential Properties

This scheme works well because it maintains a number of essential properties:

1. The first property is that direct numbers are encoded directly in the normal way, without any need to encode or decode them. This also makes for very tight fast-paths for numeric primops.
2. The second property is that all of the information needed in order to determine the type is available in the high byte. This makes it possible to perform comparisons on it, after roll with al, r12b, etc. It also make it very visible in hex-printouts.
3. The third property, because the actual representations are ordered by type, we can recognize ranges of possible types with a comparison. For example, if the high-bit is greater than or equal to the smallest possible closure value, then the result is either a thunk or a closure. Similarly, anything smaller than a pin is a number; either direct or indirect.
4. The fourth property is that each heap reference can be determined by checking a specific bit. This only works if we already know that the reference is not a direct number, but if we do know that we can do the determination with a single 'bt' instruction.
5. The fifth property, is that each type of heap reference has a different number of leading zeros, so we can use lzcnt to convert from a heap reference to an enum of all possible heap types.
6. The sixth property is that, the metadata associated with a type also lives in the second highest byte, which has the same advantages as having the type live in the top-most byte.

4.2 Operations on Heap References

4.2.1 Extracting the Raw Pointer

Converting a tagged value into a pointer just requires two shifts to zero out the high bits.

```
.macro ptr r                                    # r = PTR(r)
        shl     \r, 16                          # lsh clears high bits
        shr     \r, 16                          # rsh recovers original shape
.endm

.macro ptr_ r, from                             # r = FROM(from)
        mov     \r, 48
        bzhi    \r, \from, \r                   # mask the bottom 48 bits
.endm

.macro dref r                                   # r = *PTR(r)
        ptr     \r
        mov     \r, [\r]
.endm

.macro dref_ r from                             # r = *PTR(from)
        mov     \r, \from
        dref    \r
.endm

.macro refo r, o                               # r = PTR(r)[i]
        ptr     \r
        lea     \r, [\r + \o]
.endm

.macro refo_ r, src, o                         # r = PTR(src)[i]
        mov     \r, \src
        refo    \r, \o
.endm

.macro drefo r, o                               # r = PTR(r)[i]
        ptr     \r
        mov     \r, [\r + \o]
.endm

.macro drefo_ r, src, o                         # r = PTR(src)[i]
        mov     \r, \src
        drefo   \r, \o
.endm

.macro clzix_ r, src, i
        .set    ix, 8+(\i * 8)
        drefo_  \r, \src, ix
.endm

.macro drefi r, i                               # r = PTR(r)[i*8]
        ptr     \r
        mov     \r, [\r + 8 * \i]
.endm

.macro drefi_ r, src, i                         # r = PTR(str)[i*i]
        mov     \r, \src
        ptr     \r
        mov     \r, [\r + \i * 8]
.endm
```

4.2.2 Extracting an Type Enum

In cases where we want to use a jump-table to switch on each possible type instead of just branching, we need to convert from a heap-reference into an enum:

```
thunk=0, closure=1, law=2, pin=3, nat=4
```

The implementation works by taking advantage of the fact that each heap reference type has a unique number of zeros.

This ends up not being used very often, since there are only a few cases where we want to handle each possible type in different way.

```
.macro tytag r                                 # tytag(r):
        shl     \r, 2                          #   r <<= 2
        lzcnt   \r, \r                         #   r = clz(r)
.endm
.macro tytag_ out, from                        # tytag_(out,from):
        mov     \out, \from                    #   out = inp
        tytag   \out                           #   out = tytag(out)
.endm
```

4.2.3 Direct Numbers

In order to determine if a number is a direct atom or a heap reference, we can examine the high bit, something which the architecture has direct support for.

`jdirect` jumps to a label if the the register is a direct number, and `jheap` jumps if it is anything else.

```
.macro jdirect reg, label                       # jump if direct
        test    \reg, \reg                      # examine register
        jns     \label                          # not signed == high bit clear
.endm
.macro jheap reg, label                         # jump if indirect
        test    \reg, \reg                      # examine register
        js      \label                          # signed == high bit set
.endm
```

4.2.4 Numbers

Since direct and indirect numbers are both smaller values than everything else, as long as the high byte is smaller than the smallest byte possible for a pin, then the value is a number.

`jinat` jumps to a label if the register is a nat, `jnnat` jumps if it's not. Both clobber r12.

`ncast` casts an already-evaluated register into a nat, clobbering r11 and r12.

```
.macro jinat reg, lab                           # jump if nat
        rorx    r12, \reg, 56                   # rorx to get high byte
        cmp     r12b, 0xC4                      # smaller than a pin?
        jb      \lab                            # then jump to the label
.endm
.macro jnnat reg, lab                           # jump if not nat
        rorx    r12, \reg, 56                   # rorx to get high byte
        cmp     r12b, 0xC4                      # big enough to be a pin?
        jae     \lab                            # then jump to the label
.endm
.macro ncast reg                                # cast to nat
        xor     r11d, r11d                      # r11=0
        rorx    r12, \reg, 56                   # r12 = (reg>>56)
        cmp     r12b, 0xC4                      # if (r12 >= 0xC4)
        cmovae  \reg, r11                       #   reg=0
.endm
```

4.2.5 Thunks, App, Laws

Thunks are the largest values, so we can identify them by checking if their high bit is within a certain range.

`jithk` jumps t a label if the value is thunk, and `jnthk` jumps if it is not.

`jiclzt` jumps t a label if the value is thunk or a closure, and `jnclzt` jumps if it is anything else.

```
.macro jithk reg, lab
        rorx    r12, \reg, 56
        cmp     r12b, 0xE0
        jae     \lab
.endm
.macro jnthk reg, lab
        rorx    r12, \reg, 56
        cmp     r12b, 0xE0
        jb      \lab
.endm
.macro jiclzt reg, lab                          # Jump if closure or thunk
        rorx    r12, \reg, 56
        cmp     r12b, 0xD0
        jae     \lab
.endm
.macro jnclzt reg, lab
        rorx    r12, \reg, 56
        cmp     r12b, 0xD0
        jb      \lab
.endm
.macro jiclz reg, lab                           # Jump if closure.
        rorx    r12, \reg, 56
        cmp     r12b, 0xD0
        je      \lab
.endm
.macro jnclz reg, lab                           # Jump if not closure.
        rorx    r12, \reg, 56
        cmp     r12b, 0xD0
        jne     \lab
.endm
.macro jipin reg, lab                           # Jump if closure.
        rorx    r12, \reg, 56
        cmp     r12b, 0xC4
        je      \lab
.endm
.macro jnpin reg, lab                           # Jump if not closure.
        rorx    r12, \reg, 56
        cmp     r12b, 0xC4
        jne     \lab
.endm
.macro jilaw reg, lab                           # Jump if closure.
        rorx    r12, \reg, 56
        cmp     r12b, 0xC8
        je      \lab
.endm
.macro jnlaw reg, lab                           # Jump if not closure.
        rorx    r12, \reg, 56
        cmp     r12b, 0xC8
        jne     \lab
.endm
```

There are also slightly more efficient variants which can be used if we know that the input is not a direct number:

4.2.6 Laws

```
.macro _jiclz reg, lab # Jump if closure (assuming it is not a nat)
        bt      \reg, 60
        jc      \lab
.endm

.macro _jilaw reg, lab # Jump if law (assuming it is not a nat)
        bt      \reg, 59
        jc      \lab
.endm

.macro _jnlaw reg, lab # Jump if not law (assuming it is not a nat)
        bt      \reg, 59
        jnc     \lab
.endm
```

4.2.7 Specific Closure Shapes

Because the tag and size information of closures are available in the tag for small values, we can recognize specific closures shapes by comparing the high 16 bits to specific values.

This is especially useful when generating optimized code for functions which switch various properties of ADTs.

But it's also useful in a few particular cases within the runtime system itself, since we need to be able to do this in order to identify patterns within law bodies.

```
.macro jnkal reg, lab                           # Jump if not closure.
        mov     r12, \reg
        shr     r12, 48
        cmp     r12w, 0xD002
        jne     \lab
.endm
```

```
.macro jnquo reg, lab                           # Jump if not closure.
        mov     r12, \reg
        shr     r12, 48
        cmp     r12w, 0xD001
        jne     \lab
.endm
```

4.3 Asserts and Debugging

We are a complex virtual machine with complex internal invariants. So we have several macros which do nothing if the source was compiled with the `NDEBUG` preprocessor flag, but otherwise perform light validity checking on pointers and objects to catch invariant violations.

These checks should be light enough that they can be left in a release binary and precise enough that they will never false positive.

TODO: Actually transition everything over to .S files which do C preprocessor expansion so we can hide these on release builds.

4.3.1 Assertion Failure

We jump to here to trap in the debugger if things have gone wrong.

```
assert_fail:
        ud2
```

4.3.2 Assert Buddy List Pointer

Asserts that reg is within the buddy mmap, including the prelude to the buddy heap that includes the bucket control. This is used for asserting validity of the list pointers at runtime.

```
.macro ablptr reg                               # assert buddy list pointer
        cmp     \reg, buddy_mmap_ptr
        jb      assert_fail
        cmp     \reg, base_end_ptr
        jae     assert_fail
.endm
```

4.3.3 No 0xBADBEEFBADBEEF Values

Validates that the register does not contain the sentinel 0xBADBEEFBADBEEF value.

TODO: An invalid is kind of heavyweight. Can we just check the lower word and then jump on low word to the rest of the check somehow?

```
.macro nobeef reg
        movabs  r12, 0xBADBEEFBADBEEF
        cmp     \reg, r12
        je      assert_fail
.endm
```

5 Heap Object Layout

The word immediately preceding each heap object is a GC record which includes it's size and type. This is needed by the first-generation GC because Cheney's algorithm needs to do a linear walk of the heap.

At the moment, this size is *also* used to encode the size of numbers and closures, but in the future these will be stored separately. Because of this, the GC size is stored in bits, not words.

The GC header also has a separate type for NF closures and WHNF closures, which is used to cache Force operations to avoid duplicate work.

5.1 GC Headers

The GC header is always the word right before the one pointed at. It contains a type and a bit-size, which is always a multiple of 64 for things besides nats.

```
.macro gchdr r                                  # gcheader(r) = PTR(r)[-8]
        ptr     \r
        mov     \r, [\r - 8]
.endm

.macro gcsz hdr                                 # Size of allocation box
        add     \hdr, 16128
        shr     \hdr, 14
.endm

.macro gcsz_ reg, hdr
        mov     \reg, \hdr
        gcsz    \reg
.endm

.macro gcbits hdr
        shr     \hdr, 8
.endm
```

We define a number of wrapper macros to formalize the usage patterns for extracting data from this word in different ways.

```
.macro bitsiz r
        gchdr   \r
        gcbits  \r
.endm

.macro bitsiz_ r, from
        mov     \r, \from
        bitsiz  \r
.endm

.macro bytsiz r
        bitsiz  \r
        add     \r, 7
        shr     \r, 3
.endm

.macro bytsiz_ r, from
        mov     \r, \from
        bytsiz  \r
.endm

.macro dbitsz r
        xchg    \r, r12
        mov     \r, 64
        lzcnt   r12, r12
        sub     \r, r12
.endm

.macro dbitsz_ r, from
        lzcnt   \r, \from
        neg     \r
        add     \r, 64
.endm

.macro dbytsz_ r, from
        dbitsz_ \r, \from
        add     \r, 7
        shr     \r, 3
.endm

.macro wordsz r
        gchdr   \r
        gcsz    \r
.endm

.macro wordsz_ r, from
        mov     \r, \from
        wordsz  \r
.endm

.macro slotz_ r, from
        mov     \r, \from
        slotz   \r
.endm

.macro slotz r                                  # wordsize, assuming not a nat
        gchdr   \r
        shr     \r, 14
.endm

.macro clsz r                                   # fn(x) = slots(x)-1
        slotz   \r
        dec     \r
.endm

.macro clsz_ r, from
        mov     \r, \from
        clsz    \r
.endm
```

5.2 Extracting Data from Heap Objects

Getting the tag of a closure is just a dereference.

```
.macro clshd r
        drefo   \r, 0
.endm

.macro ix0 r
        drefo   \r, 8
.endm

.macro ix0_ r, from
        drefo_  \r, \from, 8
.endm

.macro ix1 r
        drefo   \r, 16
.endm
.macro ix1_ r, from
        drefo_  \r, \from, 16
.endm
.macro unpin r
        dref    \r
.endm
.macro unpin_ r, from
        dref_   \r, \from
.endm
.macro name r
        dref    \r
.endm
.macro name_ r, from
        dref_   \r, \from
.endm
.macro arity r
        drefo   \r, 8
.endm
.macro arity_ r, from
        drefo_  \r, \from, 8
.endm
.macro body r
        drefo   \r, 16
.endm
.macro body_ r, from
        drefo_  \r, \from, 16
.endm
```

5.3 Big Numbers

TODO

5.4 Closures

TODO

5.5 Thunks

TODO

5.6 Liquid Pins

TODO

5.7 Frozen Pins

TODO

5.8 Mega Pins

TODO

5.9 Claim and Reserve

TODO

5.10 Pending Changes

Once we add support for non-normalized pins and laws on the 1st-gen heap, we will need to have a NF flag for pins and laws as well.

TODO: explicit nat and closure size to enable in-place shrinking.

6 Allocation

6.1 mkbuffer

This creates a new bignat directly on GC2 in order to prevent it from moving around. This allows us to get pointers into nats for use during syscalls, and for casting to function pointers in order to generate code for pins and laws.

This takes a single argument: the number of bytes that we will need.

The process here is that we allocate one byte more than needed and set the high byte to 1 (to create a BAR of the right size).

`buddy_malloc` does not zero the memory, so we do that. It also does not set the actual bit-size, and we need that, so we overwrite the GC header it creates with our own.

```
.global mkbuffer
mkbuffer:                                       # mkbuffer: (rdi=size)
        mov     r12, rdi                        #   r12=bytes
        mov     rbx, 8                          #
        cmp     rdi, rbx                        #   size = max(size, 8)
        cmovb   rdi, rbx                        #
        shr     rdi, 3                          #   words = bytes/8
        inc     rdi                             #   words++
        mov     rbx, rdi                        #   rbx=words
        xor     esi, esi                        #   tag=0
        call    buddy_malloc                    #   buddy_malloc() (rax=ptr)
        mov     rsi, rax                        #   copy pointer to rsi
        mov     rcx, rbx                        #   count=words
        mov     rdi, rax                        #   dst=ptr
        xor     eax, eax                        #   value=0
        rep     stosq                           #   memcopy
        mov     byte ptr [rsi+r12], 1           #   set high byte
        mov     rax, 0x8200000000000000         #   result=NATMASK
        or      rax, rsi                        #   result |= ptr
        shl     r12, 3                          #
        inc     r12                             #   bits = (bytes*8)+1
        shl     r12, 8                          #   hdr = (bits, tag=0)
        mov     [rsi-8], r12                    #   replace header w/ correct sz
        ret                                     #   return result
```

TODO: make sure this is actually collected propertly.

TODO: make sure that GC2 correctly handles an actual bit-size, not just a word-size wrapped in a bit-size. If not, we can just provide a fake GC header as the first word within this allocation. We are going to have to do that eventually, once we include the mark and the intrinsic linked list.

6.2 mkword

`mkword` takes any 64-bit number, and returns a valid PLAN value. The result will either be a direct number, or a bignum depending on the size.

Because this is often used to finalize a result, this uses the unconventional calling convention of having both the argument and the return value in rax. Only r8 is clobbered.

```
mkword:                                         # mkword: (rax=word)
        test    rax, rax                        #   if high bit is set
        js      mkword64                        #     tailcall mkword64(word)
        ret                                     #   return
```

6.3 mkword64

`mkword64` is similar to `mkword` except that the input is required to have it's high bit set, and so the output is always a single-word bignum.

Because this is often used to finalize a result, this uses the unconventional calling convention of having both the argument and the return value in rax. Only r8 is clobbered.

Note that the GC header is set to `0x4000` which is tag=0, size=64: `0x4000 = 64<<8 + 0`. It is always this size, because all valid single-word heap-allocated nats are exactly 64-bits wide.

The pointer tag is 0x8200000000000000, which is just the bit-pattern for NAT with no metadata.

```
mkword64:                                       # mkword64: (rax=word)
        lea     r8, [r14+16]                    #   tmp = hp+2
        cmp     r8, r13                         #   if (tmp > heapend)
        ja      mkword64.gc                     #     goto gc
        xchg    r8, r14                         #   ptr=hp, hp=tmp
        mov     qword ptr [r8], 0x4000          #   gc record
        add     r8, 8                           #   ptr++
        mov     [r8], rax                       #   *ptr = word
        mov     rax, 0x8200000000000000         #   result = mask
        or      rax, r8                         #   result |= word
        ret                                     #   return result
mkword64.gc:                                    # gc:
        not     rax                             #   hide 64bit word via negation
        ppush   rdi                             #   save rdi
        mov     rdi, 2                          #   need two words
        call    gc                              #   gc()
        ppop    rdi                             #   restore rdi
        not     rax                             #   restore 64bit word
        jmp     mkword64                        #   try again
```

6.4 mkdouble

This allocates a two-word NAT, allocating a heap object where necessary.

Note that both input arguments may be in GC-unsafe states, since unpacked word64s can look like heap references to the GC. We need to be very careful to hide these from GC and we need to make sure they are cleared before we return.

This follows the unconventional calling convention where the arguments are in rax and rdx, and the return value is in rax.

```
mkdouble: # rax=lo, rdx=hi                      # mkdouble128:
        test    rdx, rdx                        #   if (rdx == 0)
        jz      mkword                          #     return mkword(rax)
mkdouble128:                                    # mkdouble128:
        lea     rsi, [r14+8*3]                  #
        cmp     rsi, r13                        #   if (heap_next > heap_end)
        ja      mkdouble128.gc                  #     then gc
        xchg    rsi, r14                        #   swap (heap_next, rsi)
        mov     [rsi+8], rax                    #   rsi[1] = low_word
        mov     [rsi+16], rdx                   #   rsi[2] = high_word
        lzcnt   rdx, rdx                        #
        mov     rax, 128                        #
        sub     rax, rdx                        #   rax = 128 - clz(high_word)
        shl     rax, 8                          #
        mov     [rsi], rax                      #   RECORD (t=nat/0 sz=bitsz)
        add     rsi, 8                          #   res = &gc_ptr[1]
        mov     rax, 0x8200000000000000         #
        or      rax, rsi                        #   rax=(res | 0x8200000000000000)
        ret                                     #   return (rax+rdx both safe)
mkdouble128.gc:                                 #
        push    rax                             #   Hide 64-bit words on c stack
        push    rdx                             #
        xor     rax, rax                        #
        xor     rdx, rdx                        #
        ppush   rdi                             #
        mov     rdi, 3                          #   needed words
        call    gc                              #
        ppop    rdi                             #
        pop     rdx                             #
        pop     rax                             #
        jmp     mkdouble128                     #
```

7 Numeric Utilities

7.1 openat

Given any PLAN value, coerces the input into a number (non-numeric values are cast to 0), and returns a pointer to the u64[] containing the number along with it's bit-width (special-casing 64 if 0).

This routine is used for numeric operations, and it solves tow problems at once:

1. Handle the edge-case of a non-numeric input being passed to a operation on natural numbers. (All PLAN jets handle this by casting non-nat value to 0).
2. Converts direct and indirect natural numbers into a uniform representation, which makes it possible to use the same code for all combinations of representations (indirect - indirect), (direct - indirect), (indirect - direct), and (direct - direct).

There is one special cases, which is a bit of a hack: When the input is zero, we actually return 64 for the bit-size. In practice, we will always use the bit-size to compute the word-size of the resulting buffer, which is always one, even if the input is zero.

```
opennat: # rdi=plan value,rsi=buffer            # openat:
        jdirect rdi, opennat.direct
opennat.indirect:
        bitsiz_ rax, rdi                        # rax = bitsize(rdi/bitnat)
        ptr_    rdx, rdi                        # rdx = ptr(rdi/ref)
        ret
opennat.direct:
        mov     [rsi], rdi
        mov     rdx, rsi                        # *tmp=word, rdx=temp
        mov     rax, 64                         # rax=64
        test    rdi, rdi
        jz      opennat.direct_zero_sized
        lzcnt   rcx, rdi
        sub     rax, rcx
opennat.direct_zero_sized:
        ret
```

7.2 unpack

Unpacks the top two items of the plan stack into the following standardized parts of a stack frame. This function has the unusual calling convention that it reads the top two items from the shadow stack, and then unpacks those items into the caller's stack frame:

Offset	Description
-8	x pointer tag
-16	x qword buffer for direct
-24	x bitsize
-32	x buffer
-40	x buffer word size
-48	y pointer tag
-56	y qword buffer for direct
-64	y bitsize
-72	y buffer
-80	y buffer word size

```
unpack:                                         # unpack:
        mov     rdi, [r15+8]                    #   arg1
        mov     [rbp-8], rdi                    #   rbp[-8] = arg1
        mov     rsi, [r15]                      #   arg2
        mov     [rbp-48], rsi                   #   rbp[-48] = arg2
        lea     rsi, [rbp-16]                   #
        call    opennat                         #
        mov     [rbp-24], rax                   #
        mov     [rbp-32], rdx                   #
        btw     rax                             #
        mov     [rbp-40], rax                   #
        mov     rdi, [rbp-48]                   #
        lea     rsi, [rbp-56]                   #
        call    opennat                         #
        mov     [rbp-64], rax                   #   rbp[-64] = arg2.bits
        mov     [rbp-72], rdx                   #
        btw     rax                             #
        mov     [rbp-80], rax                   #   rbp[-80] = arg2.words
        ret
```

8 Pinning

Pinning is our system for taking a tree of values up to the next pin boundary and compacting it into a single contiguous segment, recording metadata about

8.1 Hash Table

Because pins can be arbitrarily large, we must define data structures that can grow during pin construction to perform the size calculations, pointer mapping, and pointer deduplication. We implement a simple linear probing hash table that maps a qword to a qword, where the entire structure can be directly freed at the end.

```
typedef struct { uint64_t key, val; } entry_t;
typedef struct {
uint64_t lg;      // exponent of the power of 2 size of table
uint64_t mask;    // mask for current size to apply to hashes
uint64_t count;   // number of items, tracked for resizing.
entry_t tbl[];    // flexible array
} ht_t;
```

8.1.1 ht_create

We create a hash table by passing in the exponent of the power of two desired capacity of the table.

```
.global ht_create
ht_create:                                      # ht_create(rdi=lg):
        mov     rcx, rdi                        #   move to rcx fro shl
        mov     rsi, 1                          #   base 1
        shl     rsi, rcx                        #   size = 1 << lg
        ppush   rdi, rsi, rcx                   #   save registers
        mov     rdi, rsi                        #   bytes = size
        shl     rdi, 1                          #   bytes *= 2 (keys+vals)
        add     rdi, 3                          #   bytes += 3 (header words)
        mov     rsi, 0x008                      #   tag=Hash Table
        call    buddy_malloc                    #   buddy_malloc -> rax=buffer
        ppop    rdi, rsi, rcx                   #   restore registers
        mov     [rax], rdi                      #   h->lg = lg;
        mov     rcx, rsi                        #   mask = power of two size
        dec     rcx                             #   mask-- (now a mask of 1s)
        mov     [rax+8], rcx                    #   h->mask = size - 1
        inc     rcx                             #   restore size
        mov     qword ptr [rax+16], 0           #   h->count = 0
        mov     rsi, rax                        #   save return value
        lea     rdi, [rsi+24]                   #   rdi = beginning of kv buffer
        shl     rcx, 1                          #   rcx = size * 2
        xor     rax, rax                        #   set value to 0
        rep     stosq                           #   zero out table memory
        mov     rax, rsi                        #   restore return value
        ret                                     #   return
```

8.1.2 htkey

Given the base pointer and an index, write the key value at idx. We do things this way because it's cleaner at the call site to operate on masking the index.

```
.macro htkey base, idx, out
        mov     \out, \idx
        shl     \out, 4         # idx * 16; can't rdx*16 inside mov.
        mov     \out, [\base + \out + 24] # base->tbl[idx].key
.endm
```

8.1.3 htkpos

Given the base pointer and an index, write the pointer to the key at idx.

```
.macro htkpos base, idx, out
        mov     \out, \idx
        shl     \out, 4         # idx * 16; can't rdx*16 inside mov.
        lea     \out, [\base + \out + 24] # base->tbl[idx].key
.endm
```

8.1.4 htval

Like `htkey`, but points to the value.

```
.macro htval base, idx, out
        mov     \out, \idx
        shl     \out, 4         # idx * 16; can't rdx*16 inside mov.
        mov     \out, [\base + \out + 32] # base->tbl[idx].val
.endm
```

8.1.5 ht_has

Returns the index at which a key exists, or -1 if key isn't found.

```
.global ht_has
ht_has:                                     # ht_has(rdi=ht*,rsi=key):
        mov     rax, -1                     #   initialize hash to -1
        crc32   rax, rsi                    #   crc32 on key value as hash
        mov     r8, [rdi + 8]               #   r8 = h->mask
        and     rax, r8                     #   idx = hash & h->mask
ht_has.loop:                                # loop:
        htkey   rdi, rax, rcx               #   rcx = hdi->tbl[rax].key
        test    rcx, rcx                    #   if key is zero
        jz      ht_has.not_found            #   then item not found
        cmp     rcx, rsi                    #   if key is requested key
        je      ht_has.return               #   then found and return idx
        inc     rax                         #   idx++
        and     rax, r8                     #   use mask to loop around
        jmp     ht_has.loop                 #   loop
ht_has.not_found:                           # not_found:
        mov     rax, -1                     #   set return value to -1
ht_has.return:                              # return:
        ret                                 #   return
```

8.1.6 ht_put

Sets `key` to `value`, overwriting the current value if one is set.

When we put a value in a hash table, the first thing we have to do is make sure that the hash table is less than 70% full. If it is, we invoke the resizing behaviour, which doubles the size and rehashes everything.

```
ht_put:                                     # ht_put(rdi=ht*, rsi=key, rdx=val):
        mov     rax, [rdi + 16]             #   rax = tbl->count
        inc     rax                         #   add 1 for new count
        cvtsi2sd xmm0, rax                  #   convert to double in xmm0
        mov     rax, [rdi + 8]              #   rax = tbl->mask
        inc     rax                         #   rax is now capacity
        cvtsi2sd xmm1, rax                  #   convert to double in xmm1
        divsd   xmm0, xmm1                  #   xmm0 = (h->count++)/(h->mask++)
        movsd   xmm1, qword ptr [hash_table_load_factor]
        ucomisd xmm0, xmm1                  #   if percent < load_factr
        jbe     ht_put.has_valid_hash_table #   jump if not greater
        ppush   rdi, rsi, rdx               #   save registers
        call    ht_resize                   #   resize the table
        ppop    rdi, rsi, rdx               #   restore registers
        mov     rdi, rax                    #   rdi is now resized table
ht_put.has_valid_hash_table:                # has_valid_hash_table:
        mov     rax, -1                     #   initialize crc32
        crc32   rax, rsi                    #   hash the key
        mov     r8, [rdi + 8]               #   load mask
        and     rax, r8                     #   idx = hash & h->mask
ht_put.loop:                                # loop:
        mov     r9, rax                     #   r9 = idx
        shl     r9, 4                       #   r9 = idx * 16
        lea     r9, [rdi + r9 + 24]         #   r9 = key location
        mov     r10, [r9]                   #   r10 = key value
        test    r10, r10                    #   if key value is null
        jz      ht_put.increment            #   then increment count and store
        cmp     r10, rsi                    #   if key value is search key
        je      ht_put.store                #   then store here w/o increment
        inc     rax                         #   idx++
        and     rax, r8                     #   overflow by mask if needed
        jmp     ht_put.loop                 #   goto loop
ht_put.increment:                           # increment:
        mov     r8, [rdi + 16]              #   get current h->count
        inc     r8                          #   increment
        mov     [rdi + 16], r8              #   set updated h->count
ht_put.store:                               # store:
        mov     [r9], rsi                   #   current location = key
        mov     [r9 + 8], rdx               #   current location++ = val
        ret                                 #   return
```

We also have to store the floating point value for 0.7 in a place in rodata so it's loadable.

```
.section .rodata
.align 8
hash_table_load_factor:
        .double 0.7
.section .data
```

8.1.7 ht_resize

Resizing the hash table is incrementing the logarithmic size by 1 and then copying each item over.

```
ht_resize:                                  # ht_resize(rdi=ht*):
        push    rdi                         #   save input hash table
        mov     rdi, [rdi]                  #   new_lg = ht->lg
        inc     rdi                         #   new_lg++
        call    ht_create                   #   rax = ht_create(new_lg)
        pop     rdi                         #   restore input hash table
        mov     r8, [rax + 8]               #   newh->mask
        mov     r10, [rdi + 16]             #   oldh->count
        mov     [rax + 16], r10             #   newh->count = oldh->count
        mov     rsi, [rdi + 8]              #   get old mask as terminator
        inc     rsi                         #   rsi = old table size
        mov     rdx, 0                      #   i = 0
ht_resize.outer_loop:                       # loop:
        htkey   rdi, rdx, rcx               #   rcx = oldh->tbl[i].key
        test    rcx, rcx                    #   if key is zero
        jz      ht_resize.continue          #   then try next item
        mov     r10, -1                     #   initialize crc32 hashing
        crc32   r10, rcx                    #   hash the key
        and     r10, r8                     #   dst = crc32(0, k) & newh->mask
ht_resize.inner_loop:                       # inner_loop:
        htkey   rax, r10, r11               #   r11 = newh->tbl[dst].key
        test    r11, r11                    #   if dst's key is zero
        jz      ht_resize.inner_loop_target #   then place item here
        inc     r10                         #   dst++
        and     r10, r8                     #   dst = (dst + 1) & newh->mask
        jmp     ht_resize.inner_loop        #   retry next slot
ht_resize.inner_loop_target:                # inner_loop_target:
        htkpos  rax, r10, r11               #   r11 = &(newh->tbl[dst].key)
        htkpos  rdi, rdx, rcx               #   rcx = &(oldh->tbl[i].key)
        mov     r10, [rcx]                  #   read key
        mov     [r11], r10                  #   write key
        mov     r10, [rcx + 8]              #   read value
        mov     [r11 + 8], r10              #   write value
ht_resize.continue:                         # continue:
        inc     rdx                         #   i++
        cmp     rdx, rsi                    #   if i < size of old table
        jb      ht_resize.outer_loop        #   then lop
        push    rax                         #   save new table
        call    buddy_free                  #   free old table
        pop     rax                         #   restore new table
        ret                                 #   return
```

9 Evaluation

9.1 Macros

9.1.1 EVAL

EVAL Causes target to be evaluated, result in target, clobbers rax and r12. tosave registers are preserved

```
.macro EVAL target, tosave:vararg
        mov     rax, \target
        jnthk   rax, 777f
        ppush   \tosave
        dref_   r12, rax
        call    r12
        mov     \target, rax
        ppop    \tosave
777:
.endm
```

9.1.2 ERAX

This is a specialized version of EVAL, causes rax to be evaluated, clobbers r12. ALl of the registers in tosave are preserved

```
.macro ERAX tosave:vararg
        jnthk   rax, 777f
        ppush   \tosave
        dref_   r12, rax
        call    r12
        ppop    \tosave
777:
.endm
```

9.1.3 JRAX

JRAX evaluates and returns rax (uses a tail-call if rax is a thunk).

```
.macro JRAX
        jnthk   rax, 777f
        dref_   r12, rax
        jmp     r12
777:
        ret
.endm
```

9.1.4 FORCE

This evaluates the target to normal form.

```
.macro FORCE target, tosave:vararg
        ppush   \tosave
        mov     rax, \target
        call    force.newabi
        mov     \target, rax
        ppop    \tosave
.endm
```

9.1.5 ENAT

ENAT evaluates a register and converts it into nat, clobbing r11 and r12.

```
.macro ENAT reg, tosave:vararg
        EVAL    \reg, \tosave
        ncast   \reg
.endm
```

9.2 Thunk Executioners

9.2.1 xdone

xdone is used for thunks which have already been evaluated, it just returns the cached values, which is in the first slot.

Note that the garbage collector automatically recognizes thunks of this form, ignores any extra slots, and shrinks the result.

```
xdone:
        drefo   rax, 8
        ret
```

9.2.2 xhole

This i used for thunks which are in the process of being evaluated. This makes it possible to detect cases where an evaluation depends on it's own result.

TODO: Should GC also shrink these down to one word, and not treat any of the slots as live?

```
xhole:
        mov     rdi, 1 # stdout
        lea     rsi, [loopstr]
        mov     rdx, 9
        call    syscall_write
        mov     rsi, 1
        jmp     syscall_exit_group
loopstr: .string "<<loop>>\n"
```

9.2.3 xvar

This is an executioner for a dummy thunk which wraps another value with may also be a thunk. This is just used to handle certain edge cases that come up when implementing LETREC.

```
xvar:
        ppush   rax
        drefo   rax, 8
        ERAX
        ppop    rdi           # rdi=thunk
        ptr     rdi           # rdi=PTR(thunk)
        mov     [rdi+8], rax
        lea     rsi, [xdone]
        mov     [rdi], rsi
        ret
```

9.2.4 xhead

`xhead` is a specialized executioner for computing the Init of a closure. The size of the closure must be at least 2.

This is used during the implementation of Open primop, so that pattern matching on a large closure doesn't need to materialize the head unless we actually use it.

For example, if the input is `{xhead, 3[4 5 6]}` then the thunk will be replaced with `{xdone, 3[4 5]}` and the return value will be `3[4 5]`.

```
xhead:
        ppush   rax                         # save thunk
        drefo   rax, 8                      # rax=(a b)
        clzsz_  rcx, rax                    # n = rax.size
        ppush   rax                         # save oldclz
        dref_   rdi, rax                    # arg1 = oldclz.ptr
        lea     rsi, [rcx-1]                # arg2 = n-1
        call    closure                     # rax = newclz
        ppop    rsi                         # src = restore(oldclz)
        ptr     rsi                         # src = oldclz.ptr
        ptr_    rdi, rax                    # dst = newclz.ptr
        mov     r8b, [rsi-8]                # r8 = oldclz.heaprecord.type
        mov     [rdi-8], r8b                # newclz.heaprecord.type = r8
        rep     movsq                       # memcpy(newclz.ptr, oldclz.ptr, n)
        ppop    rdi                         # rdi = restore thunk
        ptr     rdi                         # rdi = thunk.ptr
        mov     [rdi+8], rax                # thunk[1] = newclz
        lea     rsi, [xdone]
        mov     [rdi], rsi                  # thunk[0] = xdone
        ret                                 # return newclz
```

9.2.5 xunknown

This is a wrapper around xunknownnoupdate which performs thunk update.

1. First, the thunk is updated to be a blackhole by replacing the executioner, which will cause evaluation loops to crash.
2. Then, we call xunknownnoupdate to do the actual evaluation.
3. Then we update the thunk again to by replacing the executioner with xdone and assigning the first slot to the result.

```
xunknown:
        lea     rdx, [xhole]
        ptr_    r10, rax
        mov     [r10], rdx
        ppush   rax
        call    xunknownnoupdate
        ppop    r10               # thunk
        ptr     r10
        lea     rdx, [xdone]
        mov     [r10], rdx
        mov     [r10+8], rax
        ret
```

9.2.6 xunknownnoupudate

Some TODOs:

- Branch on the size hand small sizes using registers (not stack).
- Specialized entry-points for small sizes.

```
xunknownnoupdate:
thapple:
        ptr_    r8, rax        # r8 = ptr thunk
        mov     rcx, [r8-8]    # fetch RECORD
        shr     rcx, 14        # heapsz (in words)
        dec     rcx            # size = heapsz-1 (ignoring fp)
        cmp     rcx, 2
        je      thapple1
        cmp     rcx, 3
        je      thapple2
        cmp     rcx, 4         # TODO: use a jump table (for arities up to 6)
        je      thapple3
        mov     r11, rcx
        mov     r9, rcx
        shl     r9, 3          #
        sub     r15, r9        # sp -= size (*8 for bytes)
        lea     rsi, [r8+8]    # src = &thunk[1]
        mov     rdi, r15       # dst = stack
        rep     movsq          # copy
        lea     rdi, [r11-1]
        jmp     apply          # return apply(args)
thapple1:
        mov     rax, [r8+8]
        mov     rdi, [r8+16]
        jmp     apply1
thapple2:
        mov     rax, [r8+8]
        mov     rdi, [r8+16]
        mov     rsi, [r8+24]
        jmp     apply2
thapple3:
        mov     rax, [r8+8]
        mov     rdi, [r8+16]
        mov     rsi, [r8+24]
        mov     rdx, [r8+32]
        jmp     apply3
```

10 PLAN vs XPLAN Primops

PLAN itself only offers three primops: Pin, Law, and Open.

However, the runtime itself implements XPLAN, which greatly extends this set of operations, as well as providing a number of mechanisms for interacting the the host operating system and for directly poking the process itself.

When running PLAN code, access to the lawful subset of these operations is done by doing jet matching in the online compiler, but in XPLAN these are invoked directly as primitives.

11 Numeric Primops

Numeric operations need to handle both direct numbers and indirect numbers, and they also need to evaluate their inputs if they are thunks.

The most common case for these operations is to be called with small numbers, and the architecture supports many of these operations natively. Because of this, these are generally organized as a fast path and a slow path, where the fast path attempts to perform the machine instruction with just enough extra machinery to detect the edge-cases and fallback to a more general, slower version of the routine.

Also, there is usually a general purpose entry-point which accepts any PLAN value and a specialized version that expects to be given evaluated numbers. The idea here, is that an optimizer will often be optimizing chains of numeric operations, and it can use these specialized routines to avoid all of the overhead around handling evaluation thunks and casting.

11.1 Nil

We expose a quick check for zero equality, returning one if the input value is zero.

```
opnil:                                      # opnil(rdi=val):
        EVAL    rdi                         #   evaluate
fastnil:                                    # fastnil:
        xor     eax, eax                    #   set return value to 0
        mov     edx, 1                      #   possible return value of 1
        test    rdi, rdi                    #   is val 0?
        cmovz   eax, edx                    #   set return value to 1 if 0
        ret                                 #   return
```

11.2 ToBit

The opposite of Nil is ToBit, which casts a value to 0 or 1.

```
optobit:                                    # optobit(rdi=val):
        EVAL    rdi                         #   evaluate
fasttobit:                                  # fasttobit:
        xor     eax, eax                    #   set return value to 0
        mov     edx, 1                      #   possible return value of 1
        test    rdi, rdi                    #   is val not zero?
        cmovnz  eax, edx                    #   set return value to 1 if not 0
        ret                                 #   return
```

11.3 Nat

Casts the input to a natural number.

```
opnat:                                      # opnat(rdi=n):
        mov     rax, rdi                    #   set to rax for evaluation
        ERAX                                #   evaluate rax
        ncast   rax                         #   cast result to nat
        ret                                 #   return
```

11.4 Increment

The increment logic offers a number of entry-points with different invariants.

- opinc: The actual increment primop, which takes a single argument which can be any PLAN value.
- fastinc: The same, but with the precondition that the argument is in WHNF and is a number.

  This works by using the add instruction to increment, which sets the sign bit if the high bit of the result is set. That condition handles two cases at once: the input is indirect, and the input is direct, but overflows into 64 bits. We don't have to worry about overflow here, because (-1) is not a valid PLAN value.
- slowinc: Handles the case where the result of the increment overflows into a bignum.

  The preconditions here are that rax must be rdi+1, and that the high bit of rax is set. So, if the high bit is unset on rdi, then we know that the specific value is 0x8000000000000000. Otherwise, we are incrementing a bignum.
- bufinc: Handles an increment of a bignum where it is known that there is enough space to hold the result (propagates the carry-bit).

```
.global fastinc
opinc:
        ENAT    rdi                         # evaluate+cast
fastinc:
        mov     rax, rdi                    # result = input
        add     rax, 1                      # rax++
        js      slowinc                     # if indirect, slowinc
        ret                                 # fast path: return (x+1)
slowinc:
        jdirect rdi, mkword64               # if direct(x) return mkword64(x+1)
        xor     eax, eax                    # clear invalid register state
        call    reservecopy                 # rax=ptr, rdx/sz = reservecopy(rdi)
        mov     r10, rdx                    # r10 = wordsz
        mov     rdi, rax                    # rdi = ptr
        call    bufinc                      # bufinc(ptr)
        lea     rdi, [r10+1]                # arg = wordsz+1
        jmp     claim                       # return claim(arg)
bufinc.loop:
        add     rax, 8                      # rax += 8
bufinc:
        add     qword ptr [rax], 1          # (*rax)++
        jc      bufinc.loop                 # if overflow, repeat
        ret                                 # return
```

11.5 Decrement

Some notes on bufdec:

This does an in-place decrement of a buffer.

Input: rax: pointer to the data in x.

Output: Rax is clobbered, There is no return value.

Invariant: The buffer must encode a non-zero number, otherwise this will clobber memory outside the buffer (note the lack of a size input).

```
.global fastdec
opdec:
        ENAT    rdi                         # Evaluate + cast
fastdec:
        jheap   rdi, slowdec                # if indirect, slowpath
        xor     eax, eax                    # result = 0
        dec     rdi                         # x--
        cmovns  rax, rdi                    # result = x
        ret                                 # return result
slowdec:
        call    reservecopy                 # rax=ptr rdx=sz (rdi rsi rdx)
        call    bufdec                      # (--)(rax)
        mov     rdi, rdx                    # arg = sz
        jmp     claim                       # return claim(sz)
bufdec:
        sub     qword ptr [rax], 1          # (*buf)--
        lea     rax, [rax+8]                # buf++
        jc      bufdec                      # if underflow, loop
        ret                                 # otherwise return
```

11.6 Addition

11.6.1 opadd

Add is the general purpose version which works with any PLAN input. Note that we evaluate the second argument first, since that's how the jet definition works out.

```
opadd:
        ENAT    rsi, rdi                    # evaluate y + cast (saving x)
        ENAT    rdi, rsi                    # evaluate x + cast (saving y)
        # fallthrough to fastadd
```

11.6.2 fastadd

fastadd is the fast path for addition, and it has the pre-condition that both arguments are already natural numbers. It attempts to do direct addition, but falls back to the slow path in all cases where that isn't enough. There are four potential cases here:

- Both inputs are direct, and the result is direct.
- Both inputs are direct, but result is too big.
- One of the inputs is direct, and the other is indirect.
- Both of the inputs are indirect.

We want to return in the first case and fallback in the others. In all of the other cases, either the addition will overflow, or the high bit will be set on the result.

```
.global fastadd
fastadd:                                    # rdi=x rsi=y
        mov     rax, rdi                    #   rax = x
        add     rax, rsi                    #   rax += y
        jc      slowadd                     #   overflow?
        js      fastadd.maybeoverflow       #   high bit set?
        ret                                 #   return
fastadd.maybeoverflow:                      # rax=res rdi=x rsi=y
        mov     rdx, rdi
        or      rdx, rsi                    #   if direct(x|y)
        jns     mkword64                    #     tailcall mkword64(res)
        # fallthrough to slowadd
```

11.6.3 slowadd

In this case, we know that at least one input is indirect.

We can handle both direct+indirect cases using the same code. One of the cases just swaps registers and then jumps to the other case.

In the direct+indirect case: we make a copy of the bignat (with reservecopy, call bufadd1, and then finalize the result with claim.

In the indirect+indirect case, we need to make sure that the first parameter is larger, then copy the first parameter into a result buffer, and perform the addition using bufadd.

Note that in both cases reservecopy reserves and zeros an extra word for us, which we need for the overflow case, and claim handles shrinking the result if that ends up not being used.

```
slowadd:
        xor     eax, eax                    # clear invalid register state
        jdirect rdi, slowadd.directx        # if x is direct, goto directx
        jdirect rsi, slowadd.directy        # if y is direct, goto directy
slowadd.slowpath:
        wordsz_ r8, rdi                     # r8 = x.sz
        wordsz_ r9, rsi                     # r9 = y.sz
        cmp     r8, r9
        jae     1f                          # if (y.sz > x.sz)
        xchg    rdi, rsi                    #     swap x and y
        xchg    r8, r9                      #     swap xsz, ysz
1:
        ppush   rsi                         # save y
        call    reservecopy                 # rax=buf rdx=sz
        ppop    rsi                         # restore y
        mov     rcx, r9                     # count = y.sz
        ptr_    rdi, rsi                    # input = y.buf
        call    bufadd                      # bufadd(buf, y.buf, count)
        lea     rdi, [rdx+1]                # rdi = sz+1
        jmp     claim                       # return claim(rdi)
slowadd.directx:
        xchg    rdi, rsi                    # swap x and y
slowadd.directy:
        mov     r11, rsi                    # r11=word (prevent clobber)
        call    reservecopy                 # rdi=bignat
        mov     rdi, rax                    # rdi=ptr
        mov     rsi, r11                    # rsi=word
        call    bufadd1                     # bufadd1(ptr, word)
        lea     rdi, [rdx+1]                # rdi = xsz+1
        jmp     claim                       # return claim(rdi)
```

11.6.4 bufadd

bufadd adds two bignats, mutating the first one in-place. The second argument cannot be wider than the first, and the first must contain enough space to handle an overflow bit.

```
Input: rdi = output buffer (*u64)
Input: rsi = add buffer (*u64)
Input: rdx = size of add buffer in words (u64)
Outputs: none
Clobbers: rcx and rax
```

This just loops over the words of the add buffer and adds each word to the output buffer while propagating a carry bit forward. If the carry bit is set at the end, we call `bufinc` to propagate the final bit forward.

```
bufadd:                                     # rax=dst rdi=src rcx=n
        clc                                 #   clear carry
1:                                          # loop:
        mov     r8, [rdi]                   #   load src word
        adc     [rax], r8                   #   dst[i] += src[i]+CF; sets CF
        lea     rax, [rax+8]                #   rax++
        lea     rdi, [rdi+8]                #   rdi++
        dec     rcx                         #   n--
        jnz     1b                          #   if (n != 0) goto loop
        jc      bufinc                      #   if (carry) goto bufinc(rax)
        ret                                 #   return
```

11.6.5 bufadd1

bufadd1 adds a single word to a bignat, mutating the bignat in-place. The bignat must have enough space to deal with a potential overflow bit.

```
Input: rdi: is the buffer
Input: rsi: the word to add
Outputs: none
Clobbers: rax
```

This works by just doing the add, setting rax to the next word, and then calling bufinc if the addition overflowed. Note that this has a very minimal register footprint. We take arguments in rdi and rsi, but we only clobber rax.

```
bufadd1:                                    # rdi=dst rsi=word
        add     [rdi], rsi                  # Perform the addition in-place
        lea     rax, [rdi+8]                # nx = &buffer[1]
        jc      bufinc                      # if (overflow) tailcall bufinc(nx)
        ret                                 # return
```

11.7 Subtract

11.7.1 opsub

```
opsub:
        ENAT    rdi, rsi
        ENAT    rsi, rdi
        # fallthrough to fastsub
```

11.7.2 fastsub

```
.global fastsub
fastsub:
        mov     rax, rdi
        or      rax, rsi  # Careful!  rax value is gc-unsafe in the slow path.
        js      slowsub
        xor     rdx, rdx
        sub     rdi, rsi
        cmovc   rdi, rdx
        mov     rax, rdi
        ret
```

11.7.3 slowsub

```
slowsub:
        xor     eax, eax # Clear GC-unsafe register
        jdirect rsi, slowsub.yword
        mov     r10, rdi              # stash rdi/rsi in unclobbered regs
        mov     r11, rsi
        call    compare               # compare arguments
        mov     rdi, r10
        mov     rsi, r11              # restore rdi/rsi
        cmp     rax, 2
        jne     slowsub.zero          # if not a>b, result is 0
        call    reservecopy           # rax = rBuf, rdx = a.wordSz
        mov     r10, rdx              # save size
        ptr_    rdi, r11
        wordsz_ rcx, r11
        call    bufsub                # bufsub(rBuf, PTR(b), b.wordSz)
        mov     rdi, r10              # restore size
        jmp     claim                 # return claim(resSz)
slowsub.zero:
        xor     eax, eax
        ret
slowsub.yword:
        mov     r11, rsi
        call    reservecopy
        mov     rdi, rax
        mov     rsi, r11
        call    bufsub1
        mov     rdi, rdx
        jmp     claim
```

11.7.4 bufsub1

```
bufsub1: # rdi/a:Ptr[Word] rsi/b:Word (clobbers rax)
        sub     [rdi], rsi
        jc      bufsub1.underflow
        ret
bufsub1.underflow:
        lea     rax, [rdi+8]
        jmp     bufdec
```

11.7.5 bufsub

This performs an in-place subtraction of two indirect atoms:

```
x = (x - y)

ARG rax: pointer to the data in x.
ARG rdi: pointer to the data in y.
ARG rcx: number of words in y.

Four registers are clobbered: rax, rdi, rcx, and r8.

There is no return value, just the mutation of the data behind rax.
```

There is an important precondition: x must always be larger than y.

The *actual value* of x must be larger than the value of y, not just the buffer size. This precondition enables us to avoid needing to futz around with the size of x.

Note that the `dec,jnz` sequence is equivalent to the `loop` instruction, but that is much slower on some machines.

```
bufsub:                                     # bufsub: (rax/x rdi/y rcx/yWords)
        clc                                 #   Clear carry
bufsub.loop:                                # loop:
        mov     r8, [rdi]                   #   r8 = y[i]
        sbb     [rax], r8                   #   x[i] -= r8
        lea     rdi, [rdi+8]                #
        lea     rax, [rax+8]                #
        dec     rcx                         #   i--
        jnz     bufsub.loop                 #   if (i != 0) loop
        jc      bufdec                      #   If underflow, decrement
        ret                                 #   Otherwise return.
```

11.8 Multiply

11.8.1 opmul

TODO: is this evaluation order correct?

```
opmul:
        ENAT    rdi, rsi                    #
        ENAT    rsi, rdi                    #
        # fallthrough to fastmul            #
```

11.8.2 fastmul

We don't have any special tricks for mul, except that the actual machine operation produces two words, so there is no overflow case.

We just check if either input is indirect by using OR and then checking the high bit.

```
.global fastmul
fastmul:                                    # fastmul:
        mov     rax, rdi                    #   tmp = arg1
        or      rax, rsi                    #   tmp |= arg2
        js      slowmul                     #   if either indirect, slowmul
        mov     rax, rdi                    #   a = x
        mov     rdx, rsi                    #   b = y
        mul     rdx                         #   rax, rdx = a*b
        jmp     mkdouble                    #   return mkdouble(rax, rdx)
```

11.8.3 slowmul

`slowmul` has the precondition that at least on argument is indirect.

TODO: can rawreserve be used instead of reserve?

TODO: can we avoid needing to stash things on the stack before reserve by moving things into different registers?

```
slowmul:
        xor     eax, eax                    # Clear out gc-unsafe register rax
        jdirect rdi, slowmul.directx
        jdirect rsi, slowmul.directy
slowmul.indirect:
        mov     rcx, rsi                    # rcx   = y (n)
        mov     rsi, rdi                    # rsi   = x (m)
        wordsz_ r8, rsi                     # nx/r8 = x.wordsz (msz)
        wordsz_ r9, rcx                     # ny/r9 = y.wordsz (nsz)
        lea     rdi, [r8+r9]                # rsz   = nx + ny
        push    rdi                         # save rsz
        ppush   rcx                         # save y
        call    reserve                     # Clobbers rax, rdi, rcx
        ppop    rcx                         # restore y
        ptr     rcx                         # rcx = n (y.buf)
        ptr     rsi                         # rsi = m (x.buf)
        lea     rdi, [r14+8]                # rdi = &hp[1]
        call    bufmul                      # call bufmul
        pop     rdi                         # restore rsz
        jmp     claim                       # return claim(rsz)
slowmul.directx:
        xchg    rdi, rsi
slowmul.directy:
        xor     eax, eax                    # result = 0
        cmp     rsi, 1
        ja      1f                          # if (y <= 1):
        cmove   rax, rdi                    #     if y=1, result=x
        ret                                 #     return result
1:      mov     rdx, rdi                    # rdx=xref (avoid clobber)
        wordsz  rdi
        mov     r8, rdi                     # r8=xsz   (avoid clobber)
        inc     rdi                         # reserve(wordsz(x) + 1)
        call    reserve                     # Clobbers rdi, rcx, rax
        ptr_    r9, rdx                     # r9 = xbuf
```

11.8.4 bufmul1

```
bufmul1:                                    # bufmul1:
        xor     ecx, ecx                    #   rcx/carry = 0
        xor     edi, edi                    #   rdi/i     = 0
bufmul1.loop:                               # loop:
        mov     rax, rsi                    #   lo    = yword
        mul     qword ptr [r9+8*rdi]        #   hi,lo = lo*xbuf[i]
        add     rax, rcx                    #   lo    = lo+carry
        adc     rdx, 0                      #   hi    = hi+cf
        mov     rcx, rdx                    #   carry = hi
        inc     rdi                         #   i++
        mov     [r14+8*rdi], rax            #   hp[i] = lo
        cmp     rdi, r8                     #   if (i < xsz)
        jb      bufmul1.loop                #     goto loop
        inc     rdi                         #   i++
        mov     [r14+8*rdi], rcx            #   hp[i] = carry
        jmp     claim                       #   tailcall claim(i)
```

11.8.5 bufmul

```
# Input: rdi -- out
# Input: rsi -- m (xptr)
# Input: rcx -- n (yptr)
# Input: r8  -- msz
# Input: r9  -- nsz
```

```
bufmul:
        push    r12
        push    r13
        xor     r10, r10                    # i=0
        xor     r13, r13                    # carry=0
bufmul.iloop:
        xor     r11, r11                    # j=0
bufmul.jloop:
        # clc                               # clear carry (pointless?)
        mov     rax, [rsi+8*r10]            # lo = m[i]
        mul     qword ptr [rcx+8*r11]       # hi:lo = lo*n[j]
        lea     r12, [r10+r11]              # i+j
        add     rax, r13                    # lo += carry
        adc     rdx, 0                      # hi += hi+cf
        add     [rdi+8*r12], rax            # out[i+j] += lo
        adc     rdx, 0                      # rdx += cf
        mov     r13, rdx                    # carry=hi
        inc     r11                         # j++
        cmp     r11, r9                     # if (j < nsz)
        jb      bufmul.jloop                #     continue
        inc     r12                         # (i+j)++
        add     [rdi+8*r12], r13            # out[i+j] += carry
        xor     r13, r13                    # carry=0
        inc     r10                         # i++
        cmp     r10, r8                     # if i < mSz
        jb      bufmul.iloop                #     continue outer
        pop     r13                         # pop r13
        pop     r12                         # pop r12
        ret                                 # return
```

11.9 Div

11.9.1 opdiv

The actual operation. It just evaluates+casts and calls fastdiv.

```
.global opdiv
opdiv: # rdi=x rsi=y
        ENAT    rdi, rsi
        ENAT    rsi, rdi
        # fallthrough to fastdiv
```

11.9.2 fastdiv

This is the fastpath for direct inputs.

We only have to check x, because if x is direct and y is indirect, then we can pretend that y is direct because we will still get the right result. Proof:

```
If x<y then x/y = 0
    (a property of integer division)

if direct(x) && !direct(y) then x<y
    (indirect numbers are bigger)

forall a b. if direct(a) && !direct(b), then rawword(b) > a.
    (because the raw word of b has the high bit set, but it is never
    set on direct numbers)

Therefore forall a b. if direct(a) then a/b == a/rawword(b).
```

We take advantage of this to avoid switching on the shape of b, and just always use it's raw register value.

TODO: [illegible]

```
fastdiv:                                  # fastdiv: (rdi=x, rsi=y)
        jheap   rdi, slowdiv              #   if indirect, slowdiv
        test    rsi, rsi                  #   if (y == 0)
        jne     1f                        #     then okay
        ud2                               #     else crash (divide by zero)
1:      xor     edx, edx                  #   high word = 0
        mov     rax, rdi                  #   low word = x
        div     rsi                       #   rax,rdx = x/y
        ret
```

11.9.3 slowdiv

```
# Precondition: rdi is an indirect nat.
# Precondition: rsi is a nat
# Precondition: rsi is non-zero.
#
# Case one: rsi is direct, but rdi is indirect.
#
# Case two: both rdi and rsi are indirect.
#
# In case two, if (rsi >= rdi), the result is 0.
```

```
slowdiv: # rsi=y rdi=x -> rax=(x/y)
[illegible]
```

11.9.4 divloop

```
divloop:
[illegible]
```

11.10 Left Shift

11.10.1 oplsh

```
oplsh:                                    # oplsh: (rdi=x rsi=shift)
        ENAT    rdi, rsi                  #   eval+cast x
        ENAT    rsi, rdi                  #   eval+cast shift
        # fallthrough to fastlsh
```

11.10.2 fastlsh

fastlsh uses a single branch to determine if it can use the 'lsh' operation safely. If the shift is less than the number of leading zeros on the input, then it is safe to use the fast path. This handles all of the edge-cases at once.

- If the input is indirect, it will have no leading zeros. Since zeros is not greater than anything, this always exits the fast-path.
- If the shift is indirect, then the number will be much greater than 64, the maximum number of leading zeros.
- If the shift would overflow into two words, then the shift would be greater than the number of leading zeros, exiting the fast path.

```
.global fastlsh
fastlsh:                                  # fastlsh: (rdi=x, rsi=shift)
        lzcnt   r8, rdi                   #   num high zeros
        cmp     rsi, r8                   #   if (shift >= lzcount)
        jae     slowlsh                   #     goto slowlsh
        mov     rcx, rsi                  #   count = shift
        mov     rax, rdi                  #   res = x
        shl     rax, cl                   #   res <<= x
        ret                               #   return res
```

11.10.3 slowlsh

If we fall off of the fast path, we need take a closer look at our inputs. If the input is indirect, or the `(shift >= 64)` then we are dealing with the bignum variant. Otherwise, we can the double-precision variant for a 128-bit (or less) result.

```
slowlsh:                                  # slowlsh: (rdi=x, rsi=shift)
        jheap   rdi, biglsh               #   indirect nat? slowlsh
        cmp     rsi, 64                   #   if (shift >= 64)
        jae     biglsh                    #     goto biglsh
        mov     rcx, rsi                  #   count=shift
        xor     edx, edx                  #   hi=0
        mov     rax, rdi                  #   lo=x
        shl     rax, cl                   #   rax = new lo
        shld    rdx, rdi, cl              #   rdx = new hi
        jmp     mkdouble                  #   goto mkdouble(rdx,rax)
```

11.10.4 biglsh

```
biglsh:                                   # biglsh:
        ppush   rsi, rdi                  #
        push    rbp                       #
        mov     rbp,rsp                   #
        add     rsp, -96                  #
        call    unpack                    #
        mov     rax, [rbp-48]             #   load shift
        test    rax, rax                  #   if (shift==0)
        jz      biglsh.zero               #     goto zero
        mov     rax, [rbp-64]             #   load shift.bits
        cmp     rax, 63                   #   if (shift.bits > 63)
        ja      biglsh.huge               #     goto huge
        mov     rdi, [rbp-24]             #
        add     rdi, [rbp-48]             #
        btw     rdi                       #
        mov     [rbp-88], rdi             #
        call    reserve                   #
        mov     [rbp-96], rax             #
        call    unpack                    #
        mov     rdi, [rbp-96]             #
        mov     rsi, [rbp-32]             #
        mov     rdx, [rbp-40]             #
        mov     rcx, [rbp-48]             #
        call    buflsh                    #
        mov     rdi, [rbp-88]             #
        call    claim                     #
        pdrop   2                         #
        leave                             #
        ret                               #
biglsh.zero:                              #
        pdrop   2                         #
        mov     rax, [rbp - 8]            #
        leave                             #
        ret                               #
biglsh.huge:                              #
        ud2                               #
```

11.10.5 buflsh

```
buflsh:
        push    rbx
        push    r12
        mov     r9, rcx
        shr     r9, 6
        and     rcx, 63
        test    rcx, rcx
        jz      buflsh.word_only
        test    r9, r9
        jz      buflsh.basic
        xor     r8, r8
buflsh.zero_loop:
        cmp     r8, r9
        jge     buflsh.basic
        mov     qword ptr [rdi], 0
        add     rdi, 8
        inc     r8
        jmp     buflsh.zero_loop
buflsh.basic:
        xor     r8, r8
        xor     r9, r9
        mov     rbx, 64
        sub     rbx, rcx
        mov     r12, rcx
buflsh.loop:
        cmp     r8, rdx
        jge     buflsh.finalize
        mov     r10, [rsi + r8*8]
        mov     r11, r10
        mov     rcx, r12
        shl     r11, rcx
        or      r11, r9
        mov     [rdi + r8*8], r11
        mov     r9, r10
        mov     rcx, rbx
        shr     r9, rcx
        inc     r8
        jmp     buflsh.loop
buflsh.finalize:
        cmp     r9, 0
        je      buflsh.return
        mov     [rdi + r8*8], r9
buflsh.return:
        pop     r12
        pop     rbx
        ret
buflsh.word_only:
        lea     rdi, [rdi + r9*8]
        mov     rcx, rdx
        rep     movsq
        pop     r12
        pop     rbx
        ret
```

11.11 Read a Bit

The `Test` operation reads the value of a specific bit in a nat.

The basic implementation outline is fairly simple: if it is direct, use `bt`. If it is indirect, fetch the appropriate words and then use `bt`.

However, also note that we never have to check if the index is indirect or not, because indirect values will always be out of bounds, and so will the actual register value. Pretending that the index is always direct always gives the right results.

Another thing to note is that we do not need the out-of-bound check for the direct case. Since the high bit is always zero, simply capping the index to a maximum value of 63 always gives the right result.

The indirect case could jump to `optest.direct`, but the capping logic is unnecessary there, since the computed bit-index is always in range.

```
optest:                                   # optest: rdi=i rsi=nat
        ENAT    rdi, rsi                  #   eval+cast
        ENAT    rsi, rdi                  #   eval+cast
        jheap   rsi, optest.indirect      #   if rsi.big goto indirect
optest.direct:                            # direct: rdi=i rsi=nat
        mov     rdx, 63                   #
        cmp     rdi, rdx                  #   i = min(i, 63)
        cmova   rdi, rdx                  #
optest.directknown:                       # directknown
        xor     eax, eax                  #   res=0
        bt      rsi, rdi                  #   check bit
        mov     rdx, 1                    #   1
        cmovc   rax, rdx                  #   if set, res=1
        xor     esi, esi                  #   wipe nat (in case invalid)
        ret                               #   return res
optest.indirect:                          # indirect:
        bitsiz_ rcx, rsi                  #   rcx=bits
        cmp     rdi, rcx                  #   if (i >= bits)
        jae     ret0                      #     return 0
        mov     rdx, rdi                  #
        shr     rdx, 6                    #   wordix = bitix/64
        ptr     rsi                       #   nat = PTR(nat)
        mov     rsi, [rsi+rdx*8]          #   nat = nat[wordix]
        and     rdi, 63                   #   idx = idx % 64
        jmp     optest.directknown        #   goto directknown
```

11.12 Write a Bit

The `Set` operation writes a 1 to a specific bit of the given number.

We rely on the fact that indirect bignums always have the high bit set in order to have a fast path with a single branch. This works by setting the min(i, 63) bit. If the high bit was already set (indirect case), or the result will not fit in 63 bits (i >= 63), then the high bit of the result will be set, triggering the slow path. Otherwise, we can just return this result.

If we leave the fast path, then the result is guaranteed to be indirect.

Note that doing a `bts` on registers is much faster than doing it on memory because the memory variant has atomicity guarantees which we don't need, hence the explicit load and store.

```
opset:                                    # opset: rdi=i rsi=x
        ENAT    rdi, rsi                  #   eval+cast i
        ENAT    rsi, rdi                  #   eval+cast x
fastset:                                  #
        mov     r8, 63                    #
        mov     r9, rdi                   #
        cmp     r9, r8                    #
        cmova   r9, r8                    #   tmp = min(i, 63)
        mov     rax, rsi                  #
        bts     rax, r9                   #   res = bts(x, tmp)
        test    rax, rax                  #   if (high bit set)
        js      slowset                   #     enter then slow path
        ret                               #   return result
slowset:                                  # slowset: rdi=i rsi=x
        xor     eax, eax                  #   clear rax to avoid bad gc state
        mov     r8, rdi                   #   r8/i (avoid clobber)
        mov     r9, rsi                   #   r9/x (avoid clobber)
        lea     rsi, [r8+64]
        shr     rsi, 6                    #   minsz = ((i+1)+63)/64
        mov     rdi, r9
        call    reservecopyflex           #   rax/ptr rdx/wid
        mov     rdi, r8
        shr     rdi, 6                    #   wix = i / 64
        movzx   rsi, r8b                  #   bix = i % 64
        mov     r12, [rax+8*rdi]          #   tmp = buf[wix]
        bts     r12, rsi                  #   tmp = bts(tmp, bix)
        mov     [rax+8*rdi], r12          #   buf[wix] = tmp
        xor     r12, r12                  #   clear tmp (could be invalid)
        mov     rdi, rdx
        jmp     claim                     #   return claim(wid)
```

11.13 Clear a Bit

The `Clear` operations writes a 0 to a specific bit of the given number.

For the direct case, the high bit is always zero, so clearing that has no effect. Thus, we can achive the desired result by doing `btr(word, max(63, index))`. This also works correctly if the index is indirect.

The slow path can produce a direct number iff we're looking to set the 63rd bit of a 1 word indirect number.

```
.global fastclear
opclear:                                  # opclear: rdi=i rsi=x
        ENAT    rdi, rsi                  #   eval+cast i
        ENAT    rsi, rdi                  #   eval+cast x
fastclear:                                # fastclear:
        jheap   rsi, slowclear            #   If x is indirect, slowpath
        mov     rax, rsi                  #   result = x
        mov     rsi, 63                   #   rsi=63
        cmp     rdi, rsi                  #   if (i > 63)
        cmova   rdi, rsi                  #     i=63
        btr     rax, rdi                  #   clear bit
1:      ret                               #   return result
slowclear:                                # slowclear: rdi=i rsi=x
        mov     rax, rsi                  #   result = x
        bitsiz  rsi                       #   n = bitsiz(x)
        cmp     rdi, rsi                  #   if (i >= n) return
        jae     1b
        mov     r8, rdi                   #   r8/i (avoid clobber)
        mov     r9, rax                   #   r9/x (avoid clobber)
        mov     rdi, rax
        call    reservecopy               #   rax/ptr rdx/wid
        mov     rdi, r8                   #
        shr     rdi, 6                    #   wix = i / 64
        movzx   rsi, r8b                  #   bix = i % 64
        mov     r12, [rax+8*rdi]          #   tmp = buf[wix]
        btr     r12, rsi                  #   tmp = btr(tmp, bix)
        mov     [rax+8*rdi], r12          #   buf[wix] = tmp
        xor     r12, r12                  #   clear tmp (could be invalid)
        mov     rdi, rdx                  #
        jmp     claim                     #   return claim(wid)
```

11.14 Read a Byte

```
opload8:
        ENAT    rdi, rsi
        ENAT    rsi, rdi
fastload8:
        mov     rcx, rdi # TODO: reallocate registers
        mov     rax, rsi
        jheap   rax, fastload8.indirect
        cmp     rcx, 8
        jae     fastload8.zero
        shl     rcx, 3              # bits to drop
        shr     rax, rcx
        movzx   rax, al
        ret
fastload8.indirect:
        bitsiz_ r10, rax            # r10 = bitsize(rax/bignat)
        add     r10, 7
        shr     r10, 3              # bytesz
        cmp     rcx, r10            # if (i >= sz) return 0
        jae     fastload8.zero
        ptr     rax                 # rax = natptr
        mov     al, [rax + rcx]     # rax = natptr[i]
        movzx   rax, al
        ret
fastload8.zero:
        xor     rax, rax
        ret
```

11.15 Write a Byte

The Store8 operation sets a specific byte of a number.

- (Wipe x u a) | Sub a (Lsh (Cut u a) u)
- (Edit u w x a) | Add (Wipe u w a) (Lsh (Trunc w c) u)
- (Store w v x a) | Edit mul8+w mul8+v x a
- (Store8 x v a) | Store x 1 v a

Just do this in a simple way:

- Calculate the minimum size (min = 1+(i/8)).
- Do a reservecopybuz.
- Write the byte.
- Claim the result.

This will correctly handle direct numbers, though not in the most efficient way possible.

```
.global fastStore8
opstore8:                                  # opstore8: rdi/i rsi/b rdx/a
        ENAT    rdi, rsi, rdx              #   eval+cast index
        ENAT    rsi, rdi, rdx              #   eval+cast word
        ENAT    rdx, rdi, rsi              #   eval+cast nat
fastStore8:                                # fastStore8:
        mov     r8, rdi                    #   r8/i
        mov     r9, rsi                    #   r9/b
        mov     rdi, rdx                   #   rdi/a
        mov     rsi, r8                    #   x = n
        shr     rsi, 3                     #
        inc     rsi                        #   y = (i/8)+1
        call    reservecopyfillsz          #   buf, wid = reservecopyfill(x,y)
        jdirect r9, 1f                     #   if (indirect(r9))
        dref    r9                         #     r9 = r9.load()
1:      mov     [rax+r8], r9b              #   buf[i] = (b8) b
        xor     r9w, r9w                   #   clear r9
        mov     rdi, rax                   #   size = wid
        jmp     claim                      #   return claim(wid)
```

11.16 Write a 64-Bit Word

The Store64 operation sets a specific word of a number.

This will correctly handle direct numbers, though not in the most efficient way possible.

In order to calculate the minimum word-size of the result, we need to be careful to correctly handle the case where the write is not word aligned. Here is the formula with some examples:

```
last_byte = index+7
min_words = (last_byte/8)+1
formula   = ((index+7)/8)+1

0 -> 1 = (0+7)/8) ; at index 0, we need one word
1 -> 2 = (1+7)/8) ; at index 1, we need two
8 -> 2 = (8+7)/8) ; at index 8, still only two
9 -> 3 = (9+7)/8) ; at index 9, we need three
```

```
.global fastStore64
opstore64:                                 # opstore64: rdi/i rsi/w rdx/a
        ENAT    rdi, rsi, rdx              #   eval+cast index
        ENAT    rsi, rdi, rdx              #   eval+cast word
        ENAT    rdx, rdi, rsi              #   eval+cast nat
fastStore64:                               # fastStore64:
        mov     r8, rdi                    #   r8/i
        mov     r9, rsi                    #   r9/w
        mov     rdi, rdx                   #   rdi/a
        lea     rsi, [r8+7]                #
        shr     rsi, 3                     #
        inc     rsi                        #   y = ((i+7)/8)+1
        call    reservecopyfillsz          #   buf, wid = reservecopyfill(x,y)
        jdirect r9, 1f                     #   if (indirect(r9))
        dref    r9                         #     r9 = r9.load()
1:      mov     [rax+r8], r9               #   buf[i] = word
        xor     r9, r9                     #   clear r9
        mov     rdi, rax                   #   size = wid
        jmp     claim                      #   return claim(wid)
```

11.17 Write a 64-Bit Word In-Place

TODO: Consider changing the semantics of store64Uniq so that it will refuse to grow the result (only keeping bits which are in range).

11.18 Read a Byte Range

Load is just reserve, memcpy, claim except that you need to cap the size of the output of it if calls.

Load reads a byte-range of a Nat into another Nat.

Some notes on the handling of edge cases:

- If the offset is indirect, then the actual register value will have the high bit set, which is a value that is much larger than the biggest possible Nat, producing a zero result. Since this is the correct result for all possible inputs, we don't have to specifically handle this case.
- Similarly, since we only read bytes which actually exist in the input nat, a massive result width is just fine. Using the same trick as the last point, we can treat indirect words as indirect words, since the register value will always be large enough so that the read-size needs to be capped to what's actually available.

Some notes on edge-cases in the direct path:

- bzhi preserves all bits if given a count >= 64, which is the right behavior. However, we still need to cap the width to a maximum of 8 bytes, otherwise the conversion to bits could overflow.
- Large offsets must be explicitly handled for the same reason as well (or else the conversion from bits to bytes could overflow).

shr also has weird behavior if we were ever to shift with a count >= 64, but the above bounds check means that we don't have to worry about that, since the largest bit-offset we will be shifting is 56.

```
.global opload
opload:                                    # opload: rdi=offset rsi=width, rdx=nat
        ENAT    rdi, rsi, rdx              #   eval+cast index
        ENAT    rsi, rdi, rdx              #   eval+cast word
        ENAT    rdx, rdi, rsi              #   eval+cast nat
        # fallthrough to fastload
```

```
.global fastload
fastload:                                  # fastload: rdi=off rsi=wid rdx=num
        jheap   rdx, fastload.indirect
fastload.direct:
        mov     rax, 8
        cmp     rdi, rax                   # if (offset >= 8)
        jae     ret0                       #   return 0 (avoids lsh overflow)
        cmp     rsi, rax
        cmova   rsi, rax                   # width = min(width, 8)  (avoids lsh overflow)
        shl     rdi, 3                     # offset bytes to bits
        shl     rsi, 3                     # width bytes to bits
        mov     rax, rdx                   # result = num
        shrx    rax, rax, rdi              # result >>= offset
        bzhi    rax, rax, rsi              #   extract first n bits from res
        ret
fastload.indirect:
        # Calculate capped size
        wordsz_ rax, rdx                   # rax = words(rdx)
        shl     rax, 3                     # bytes = words*8
        sub     rax, rdi                   # rax=size(rdx) after offset
        jbe     ret0                       # if (offset >= size) return 0
        cmp     rsi, rax
        cmovb   rax, rsi                   # rax=min(size(inp), width)

        ppush   rdi, rsi, rdx
        mov     rdi, rax                   # rdi=width (in bytes)
        add     rdi, 7
        shr     rdi, 3                     # rdi=width (in words)
        call    reserve
        ppop    rdi, rsi, rdx

        xchg    rdx, rsi                   # rdx/width rsi/nat
        mov     rcx, rdx                   # count = width
        ptr     rsi                        # src = PTR(input) nat
        add     rsi, rdi                   # src += offset
        mov     rdi, rax                   # dst = rax from reserve
        rep     movsb                      # memcpy

        mov     rdi, rdx                   # rdi = bytesize
        add     rdi, 7
        shr     rdi, 3                     # rdi=width (in words)
        jmp     claim
```

11.19 Fast Copy of a Byte-Range

Snore copies a range of an input nat into a range of an output nat.

This is a load-bearing operation, because it is used in many places for building up bigger numbers by combining many different small and large numbers. For example, in serialization, printing, and operations on packed arrays. Because of this, each of the four cases (direct/indirect * input/output) have fairly well-optimized code paths.

11.19.1 opsnore

As always, the fully general entry-point needs to evaluate and cast all of the inputs.

```
opsnore:                                   # opsnore:
        ENAT    rdi, rsi, rdx, rcx, r8     #   eval+cast rdi/iof
        ENAT    rsi, rdi, rdx, rcx, r8     #   eval+cast rsi/bof
        ENAT    rdx, rdi, rsi, rcx, r8     #   eval+cast rdx/wid
        ENAT    rcx, rdi, rsi, rdx, r8     #   eval+cast rcx/inp
        ENAT    r8, rcx, rdi, rsi, rdx     #   eval+cast r8/buf
        # fallthrough to fastsnore
```

11.19.2 fastsnore

Here, we know that all of our inputs are evaluated numbers.

First we check if the output buffer is direct or indirect. If it is direct, we calculate the bar width of the output minus the output offset and constrain the copy-width to be no larger than that.

And then we branch to handle the case of a copy from a small input to a small output.

```
.global fastsnore
fastsnore:                                 # fastsnore:
        mov     rax, r8                    #   result = buffer
        jheap   r8, bufsnore               #   indirect buffer -> bufsnore
        dbytsz_ r10, r8                    #   r10 = buffer size
        dec     r10                        #   r10 = size of writable portion
        sub0    r10, rsi                   #   r10 = writable size after offset
        cap     rdx, r10                   #   rdx/wid = safe write-size
        jheap   rcx, funnelsnore           #   indirect input -> funnelsnore
        # fallthrough to smolsnore
```

11.19.3 smolsnore

When we are copying from a small input into a small buffer we don't need any branches or memory operations. Out-of-bounds slices are handled using a conditional move at the end.

First, we calculate the size of the input and constrain the copy-width to always be in-bounds (ignoring trailing zeros).

Next, we use `shrx & bzhi` to extract the relevant bits from the input.

Then we use `bzhi & shlx & andn` to zero out the destination bits from the output.

And finally, we use `shlx & or` in order to write the input bits into the output.

If the copy-width was zero, some of these computations might have produced non-sense, but that is harmless. We simply ignore what the result of this computation and return the result unchanged.

```
smolsnore:                                 # smolsnore:
        dbytsz_ r10, rcx                   #   r10 = input size
        sub0    r10, rdi                   #   r10 = readable size after offset
        cap     rdx, r10                   #   rdx = safe read-size
smolsnore.bounded:                         # bounded: (used by funnelsnore)
        shl     rdx, 3                     #   rdx = wid in bits
        shl     rdi, 3                     #   rdi = iof in bits
        shl     rsi, 3                     #   rsi = bof in bits
        shrx    rcx, rcx, rdi              #   shr to apply input offset
        bzhi    rcx, rcx, rdx              #   bzhi to extract relevant bits
        mov     r9, -1                     #   r9/mask = -1
        bzhi    r9, r9, rdx                #   r9/mask = (1 << wid) - 1
        shlx    r9, r9, rsi                #   r9/mask = mask << bof
        andn    r9, r9, rax                #   tmp = buf & ~mask
        shlx    rcx, rcx, rsi              #   input <<= bof
        or      r9, rcx                    #   tmp |= (input << bof)
        test    rdx, rdx                   #   if (wid != 0)
        cmovnz  rax, r9                    #     res = tmp
        ret                                #   return res
```

11.19.4 funnelsnore

Copying from an indirect input into a direct buffer is the same as copying from a direct input into a direct buffer except that we have to load the relevant chunk memory first.

We need to be careful not to read past the end of the input, but it is safe to read a few bytes before the beginning of the input (since there is always a GC header there).

The first thing this code does is to cap the size to always be in bounds, we have to do this even though the destination is smaller than the input because the slice could be at the end of the input.

Then, we need to do an early return if the resulting width is zero, because otherwise we might do an out-of-bounds read (given a huge input offset, for example).

Then, get a pointer into the input data, advance it by the input offset, and the slide it back so that when we do the read, the sliced bytes are the high bytes of the result.

Then we shift this back, removing all of the irreevant bytes from the input, and we set the input offset to zero.

At this point, we have all of the input data in the low bits of a direct words. And we know that everything is in-bounds for both input and output. Here, the logic is the same as the direct+direct case, so we just jump into that code.

```
        # rdi/iof rsi/bof rdx/wid rcx/inp rax/r8/buf
        # buf is direct, inp is indirect
funnelsnore:
        bytsiz_ r9, rcx                    #   max  = inp.bytes
        sub0    r9, rdi                    #   max -= input-offset
        cap     rdx, r9                    #   wid = min(wid,max)
        test    rdx, rdx                   #   if (wid==0)
        jz      funnelsnore.ret            #     return buf
        refo    rcx, rdi                   #   p = PTR(inp)+iof
        mov     r9, 8                      #   tmp = 8
        sub     r9, rdx                    #   tmp = 8 - wid
        sub     rcx, r9                    #   p -= tmp
        mov     rcx, [rcx]                 #   inp = *p
        shl     r9, 3                      #   tmp *= 8 (bits)
        shrx    rcx, rcx, r9               #   inp >>= tmp (slide back)
        xor     edi, edi                   #   iof = 0
        jmp     smolsnore.bounded          #   goto smolsnore.bounded
funnelsnore.ret:
        ret
```

11.19.5 bufsnore

bufsnore handles the cases where the output buffer is indirect.

If the input is direct, that is handled by `pokesnore`, otherwise this is just a memcopy after the width has been capped to be in-range and the offsets have been applied to the input and the output.

We don't have to do anything special in the case of an out-of-bounds read and write, because this will result in a zero copy-width which will mean that the memcopy does nothing.

```
        # rdi/iof rsi/bof rdx/wid rcx/inp rax/r8/buf
        # buf is indirect
bufsnore:                                  # bufsnore:
        bytsiz_ r10, r8                    #   max  = buffer byte-size
        dec     r10                        #   max  = writeable size
        sub0    r10, rsi                   #   max -= buffer-offset
        cap     rdx, r10                   #   constrain wid
        jdirect rcx, pokesnore             #   indirect input -> bigsnore
        bytsiz_ r10, rcx                   #   max  = inp.bytes
        sub0    r10, rdi                   #   max -= input-offset
        cap     rdx, r10                   #   constrain width to input size
        ptr     rcx                        #
        add     rcx, rdi                   #   rcx = PTR(inp)+iof
        ptr     r8                         #
        add     r8, rsi                    #   r8 = PTR(buf)+bof
        mov     rdi, r8                    #   dst = r8
        mov     rsi, rcx                   #   src = rcx
        mov     rcx, rdx                   #   cnt = wid
        rep     movsb                      #   memcpy
        ret                                #   return buf
```

11.19.6 pokesnore

When we are copying a slice of a direct atom into an indirect atom, we can do this using a single 64-bit load followed by a single 64-bit store, and we only have one predictable branch to handle the case where the input slice is completely out-of-bounds.

We have to be careful not to read past the *end* of the nat, because it is possible that it is at the end of a memory region. However, every nat has a 64-bit GC header, so it is always safe to read a few bytes *before* the slice.

Once we constain the slice size so that it fits within the input and output, a non-zero width guarentees that the read will be in-bounds, so we can eliminate that case by returning immediatly if it is zero.

Next, we shift the pointer backwards, so that we read a word where all of the bytes that we want to replaces are the high bytes of the word.

We then zero out these bytes with `bzhi`, shift the input into the appropriate bytes, and then combine the to with an `or`, and write the word back out to the same location that we loaded it from.

This is much simpler and faster than other approaches because we don't need to branch on the size, and we don't need a loop.

```
        # rdi/iof rsi/bof rdx/wid rcx/inp rax/r8/buf
        # buf is indirect, inp is direct
pokesnore:                                 # pokesnore:
        dbytsz_ r9, rcx                    #   tmp  = inp.bytes
        sub0    r9, rdi                    #   tmp -= iof
        cap     rdx, r9                    #   wid = min(wid, tmp)
        test    rdx, rdx                   #   if (wid==0)
        jz      pokesnore.ret              #     return buf
        shl     rdi, 3                     #   iof*8
        shrx    rcx, rcx, rdi              #   inp >>= (iof*8)
        refo    r8, rsi                    #   p = PTR(buf)+bof
        mov     r9, 8                      #
        sub     r9, rdx                    #   tmp = [8-wid]
        sub     r8, r9                     #   p -= tmp
        mov     r10, [r8]                  #   wor = *p
        shl     r9, 3                      #   tmp <<= 3 (bit clif)
        bzhi    r10, r10, r9               #   clear high bits from wor
        shlx    rcx, rcx, r9               #   inp <<= tmp (slide position)
        or      r10, rcx                   #   wor |= inp
        mov     [r8], r10                  #   *p = wor
pokesnore.ret:                             # pokesnore.ret:
        ret                                #   return buf
```

12 Closure Primops

12.1 Closure Size

```
opsz:                                      # opsz: rdi=arg
        EVAL    rdi                        #   eval arg
fastsz:                                    # fastsz:
        xor     rax, rax                   #   res = 0
        jnclz   rdi, 1f                    #   if !isapp(arg) return
        clzsz_  rax, rdi                   #   res = arg.sz()
1:      ret                                #   return
```

12.2 Closure Head

```
ophd:                                      # ophd: rdi=arg
        mov     rax, rdi                   #   res = input
        ERAX                               #   eval res
        jnclzt  rax, 1f                    #   if (!isapp(res)) return
        dref    rax                        #   res = res.hd()
1:      ret                                #   return
```

12.3 Last: Final Element

Because of the way the formalism is defined, getting the last element of a closure is a core operation.

Implementing this is easy. Since all closures have at least on parameter, the only edge-case is the cases where we are passed something besides a closure.

```
oplast:                                    # oplast: (rdi=thunk)
        EVAL    rdi                        #   eval rdi
        jnclz   rdi, ret0                  #   if (!rdi.isclz) return 0
        slots_  rcx, rdi                   #   get slots (sz+1)
        ptr     rdi                        #   p = PTR(rdi)
        mov     rax, [rdi+8*rcx-8]         #   res = p[slots-1]
        JRAX                               #   return eval(res)
```

12.4 Init: Drop Final Element

There are two cases. If given a single-element closure, return the head. Otherwise copy the closure to create a new one with one-fewer arguments.

```
opinit:                                    # opinit: (rdi=thunk)
        EVAL    rdi                        #   eval rdi
        jnclz   rdi, ret0                  #   if (!rdi.isclz) return 0
        dref_   rax, rdi                   #   hd = rdi.hd()
        clzsz_  rcx, rdi                   #   n = rdi.size
        cmp     rcx, 1                     #   if (n != 1)
        jne     opinit.copy                #     goto copy
        ret                                #   return hd
opinit.copy:                               # copy: rcx/n rdi/clz rax/hd
        ppush   rdi                        #   save oldclz
        mov     rdi, rax                   #   arg1 = hd
        lea     rsi, [rcx-1]               #   arg2 = n-1
        call    closure                    #   rax = newclz
        ppop    rsi                        #   src = restore(oldclz)
        ptr     rsi                        #   src = oldclz.ptr
        ptr_    rdi, rax                   #   dst = newclz.ptr
        mov     r8b, [rsi-8]               #   r8 = oldclz.heaprecord.type
        mov     [rdi-8], r8b               #   newclz.heaprecord.type = r8
        rep     movsq                      #   memcpy(newclz.ptr, oldclz.ptr, n)
        ret                                #   return newclz
```

12.5 Row: Closure from List

Given a head, a size, and a list, construct a row from the list.

```
Row 3 4 [1 [2 [3 [4 0]]]]  ==>  3[1 2 3 4]
```

This operation is designed to be as operationally simple as possible, which is why the size is explicit and the head is cast to a nat. Without a known size, we need to `realloc()`. If the head might be a function or closure we would need to deal with a bunch of edge-cases. The operation behavior it thus:

```
hed = NAT(hed)
row = mkrow(hed, n)
for (int i=0; i<n; i++):
    eval list
    row[i] = Ix0(list)
    list = Ix1(list)
return row
```

Note that Ix1 returns 0 for non-closures, so if the list is smaller than the size, the remaining elements are zeros. Since we have to zero-initialize the memory in order to maintain GC invariants, we take advantage of this to simply break out of the loop if given a list which is too slow.

```
oprow:                                     # oprow:
        ENAT    rdi, rsi, rdx              #   eval+cast hd
        ENAT    rsi, rdi, rdx              #   eval+cast n
        mov     rax, rdi                   #   rax/hd
        test    rsi, rsi                   #   if (n == 0)
        jz      oprow.ret                  #     return hd
        call    closure                    #   row = closure(hd, sz)
        mov     r8, rax                    #   r8/row
        mov     r9, rdi                    #   r9/hd
        xor     eax, eax                   #   rax = 0
        ptr_    rdi, r8                    #   rdi = PTR(row)
        mov     [rdi], r9                  #   row.hd = hd
        add     rdi, 8                     #   rdi = &row.slot(0)
        mov     rcx, rsi                   #   count = n
        rep     stosq                      #   memset
        xor     ecx, ecx                   #   i = 0
        mov     rdi, rdx                   #   rdi/list
        mov     rdx, rsi                   #   rdx/n
```

```
oprow.loop:                                 # loop: r8=row rdx=n rcx=i rdi=list
        cmp     rcx, rdx                    #   if (i >= n)
        jae     oprow.break                 #     break
        EVAL    rdi, rdx, rcx, r8           #   evaluate list
        jnclz   rdi, oprow.break            #   if notapp(list) break;
        drefo_  rax, rdi, 8                 #   res = rdi.ix(0)
        ptr_    r9, r8                      #   r9      = PTR(row)
        mov     [r9 + rcx*8+8], rax         #   r9[i+i] = item
        slots_  rax, rdi                    #   get gcsz
        cmp     rax, 3                      #   if (gcsz < 3)
        jb      oprow.break                 #     break
        drefo   rdi, 16                     #   list = list.ix(1)
        inc     rcx                         #   i++
        jmp     oprow.loop                  #   continue
oprow.break:                                # break:
        mov     rax, r8                     #   res = buf
oprow.ret:                                  # ret:
        ret                                 #   return res
```

13 Effectful Primops

13.1 PeekOp

TODO: there is no point in having an offset here, the caller can do the add themselves.

```
PeekOp:                                     # rdi=[bufptr off sz]
        mov     rsi, 3                      #
        call    unpackop.sized              #
        ENAT    rdi, rsi, rdx               #   rdi = c dst pointer as nat
        ENAT    rsi, rdi, rdx               #   rsi = offset inside rdi
        ENAT    rdx, rdi, rsi               #   rdx = length in bytes to copy
        # fallthrough to fastpeekop
```

13.1.1 fastpeekop

ARG rdi = c src pointer as nat
ARG rsi = offset inside rdi
ARG rdx = length in bytes to copy

```
.global fastpeekop
fastpeekop:                                 # fastpeekop: rdi=src rsi=off rdx=n
        add     rdi, rsi                    #   pointless offset logic
        lea     rcx, [rdx+8]                #
        shr     rcx, 3                      #   words = (bytes+8)/8
        ppush   rdi, rdx, rcx               #   save dest/bytes/words
        mov     rdi, rcx                    #
        call    reserve                     #   rax=buf=reserve(words)
        ppop    rsi, rcx, r8                #   rsi=ptr rcx=bytes r8=words
        mov     byte ptr [rax+rcx], 1       #   set high 1 byte
        mov     rdi, rax                    #   dst=buf
        rep     movsb                       #   copy bytes (src=ptr rcx=bytes)
        mov     rdi, r8                     #   rdi=words
        jmp     claim                       #   return claim(words)
```

13.2 PokeOp

And here is the actual primop which unpacks, evaluates, and casts all of the inputs.

TODO: the offset should be from the nat, instead of into the destination. The caller can trivially add to the pointer before they call us, but they can't easily get a pointer into the middle of a nat (we can).

```
PokeOp:                                     # PokeOp:
        mov     rsi, 4                      #   input = 0[ptr nat off count]
        call    unpackop.sized              #   load 4 operands
        ENAT    rdi, rsi, rdx, rcx          #   rdi = c dst pointer as nat
        ENAT    rsi, rdi, rdx, rcx          #   rsi = source nat
        ENAT    rdx, rdi, rsi, rcx          #   rdx = offset inside bufAddr
        ENAT    rcx, rdi, rsi, rdx          #   rcx = bytes to copy
        # fallthrough to fastpokeop
```

13.2.1 fastpokeop

And here's the actual routine, minus the actualy copying logic.

```
.global fastpokeop
fastpokeop:                                 # fastpokeop:
        ppush   rbx, rbp                    #   save rbx, rbp
        add     rdi, rdx                    #   dst = bufAddr + offset
        mov     rbx, rdi                    #   rbx=ptr
        mov     rdi, rsi                    #   arg1=src
        call    fastbytsz                   #   rax=bytes (clobbers r8)
        mov     rbp, rcx                    #   rbp=extras
        sub0    rbp, rax                    #   extra=(n-sz) (floor to 0)
        sub     rcx, rbp                    #   count -= extras
        mov     rdi, rbx                    #   dst=ptr
        add     rbx, rcx                    #   rbx=tailptr = ptr+count
        call    pokeraw                     #   pokeraw()
        mov     rdi, rbx                    #   dst=tailptr
        mov     rcx, rbp                    #   cnt=extras
        xor     eax, eax                    #   val=0
        rep     stosb                       #   memset(dst,cnt,0)
        ppop    rbx, rbp                    #   restore rbx/rbp
        ret                                 #   return 0
```

13.2.2 pokeraw

And here's a simple subroutine which just copies n bytes from a macro into a destination address.

ARG rdi=ptr (clobbered)
ARG rsi=nat (clobbered)
ARG rcx=count (clobbered)

```
pokeraw:                                    # pokeraw:
        jdirect rsi, pokeraw.direct         #   if direct, then goto direct()
pokeraw.indirect:                           # indirect:
        ptr     rsi                         #   src=ptr(nat)
        rep     movsb                       #   memcopy(dst=ptr,str,n=rcx)
        ret                                 #   return
pokeraw.direct:                             # direct:
        ppush   rsi                         #   write to memory
        mov     rsi, r15                    #   src = sp (addr of word)
        rep     movsb                       #   movsb(dst=ptr,src,n=rcx)
        ppop    rsi                         #   remove from stack
        ret                                 #   return
```

14 Heaps and Process Layout

At the highest level, the PLAN runtime manages two regions:

1. A heap which is managed using a buddy allocator. The per-thread heaps, other memory allocated by the interpreter and all of the superseded pins.
2. In binaries created by bust, a file backed persistence heap which is stored in the program binary itself, after the ELF region of the executable and which is unmapped on startup.

This memory is garbage collected in different ways:

1. Each per-thread heap is a first generation, and is collected with a thread-local moving GC using Cheney's algorithm. A new per-thread heap is allocated from the buddy heap, live values are copied to it, and the old heap is explicitly freed from the buddy heap.
2. The buddy allocator heap is also garbage collected. On a collection of this second generation, each thread submits a list of roots to the buddy allocator, including interpreter objects like the heap and any pin objects referenced by the stack or the first generation heap.
3. The persistence file is also garbage collected. On a collection of this third generation, live regions are calculated, and then we request the kernel to punch a hole on pages between the live regions.

Right now, this collection only happens offline, but we hope to make this online soon.

15 The Per Thread Heap

Each thread has its own heap, where the heap itself is allocated from the main buddy allocation arena.

15.1 Cheney Sizes

The Cheney heap can be in only one of a set of pre-chosen heap sizes. We maintain a mapping from "heap rank" to sizes. The first 20 entries are the Fibonacci sequence starting with 12 and 38, representing the start of the ramp up phase of the Cheney heap. After the first 20 entries, we switch to 20% growth per term up to a heap of 583 megabytes, at which you should just crash. We allocate space for this table:

```
.set HEAP_MAX_RANK,     51

.global heap_sizes
.align 8
heap_sizes:
        .zero HEAP_MAX_RANK * 4
```

15.1.1 build_heap_table

When on startup, we must build a table of the appropriate heap bucket sizes. These are the sizes that we allocate during Cheney collection.

We follow the precedent of the Erlang BEAM VM by using our heap allocations by a modified Fibonacci sequence, starting in the same place: 12 and 38 words. Also following BEAM, we switch from doing Fibonacci to doing a straight multiplication when Fibonacci grows too quickly.

```
[illegible]
```

15.2 Cheney State

Each worker thread (currently only one) maintains information about its own heap.

```
.global heap_addr, mapd_size, heap_size, heap_rank
heap_addr:      .quad 0
mapd_size:      .quad 0
heap_size:      .quad 0
heap_rank:      .quad 0
```

15.3 Cheney Collection

Each thread has its own Cheney heap.

15.4 asmgc

This is the core loop that drives garbage collection. It sets up the current Cheney heap in rbx/rbp, calculates how large the new heap should be, finds the needed allocation, allocates the new heap, drives the copying, and then calculates the target size to run the next collection.

Most of the work is in that final part:

- If the sum of the surviving words and the needed words is less than 25% of capacity, then shrink the target heap rank by one.
- If the sum of the surviving words and the needed words are over 50% of capacity of the current rank, then grow the target heap rank by at least one, increasing up until the sum fits, since the program could ask for much more memory than the current size.
- Otherwise, we're sized correctly between 25% and 50% of capacity and we keep the same target.

```
[illegible]
        jmp     asmgc.done                  #   completed size setting
asmgc.shrink:                               # shrink rank:
        ## TODO: Understand where Erlang BEAM sets "bigness" and how
        ## to translate that to our world. 6 is a shot in the dark.
        cmp     r9d, 6                      #   if (rank < 6) (minimum)
        jb      asmgc.keeprank              #   then keep current rank
        dec     r9d                         #   decrease rank
        mov     qword ptr [heap_rank], r9   #   set the decreased rank.
        mov     r8d, [heap_sizes+r9d*4]     #   get shrunk rank size
        mov     qword ptr [heap_size], r8   #   set shrunk size
        jmp     asmgc.done                  #   goto done
asmgc.grow:                                 # grow rank:
        inc     r9d                         #   rank++
        mov     r8d, [heap_sizes+r9d*4]     #   get increased rank size
        cmp     rax, r8                     #   if (used size > rank++ size)
        ja      asmgc.grow                  #   then increase again
        mov     qword ptr [heap_rank], r9   #   set the increased rank.
        mov     qword ptr [heap_size], r8   #   set grown size.
asmgc.done:                                 # done:
        mov     r13, [heap_addr]            #   heap_end = heap_addr
        lea     r13, [r13+8*r8]             #   heap_end = &heap_addr[heap_size]
        ret                                 #   return
asmgc.toobig:                               # too big:
        ud2                                 #   cannot fulfill this request
```

16 The Buddy Allocator

The runtime's main allocator is a buddy allocator. We choose the buddy allocator because it's well understood and simple to implement. It is also relatively easy to extend with garbage collection.

In a buddy allocator, the memory arena is represented as a binary tree where a leaf node represents an allocation. At start, there's a single leaf node that represents the entire arena. The high level idea is that if you ask for a small 4k allocation, this one large node will split to an inner node with two unallocated leaf nodes, and will do this recursively until it has tightly sized unallocated node which it hands to the user.

16.1 Buddy Configuration

Our allocator sizes are configurable, and there are two main tunables:

```
.global min_alloc_log2, max_alloc_log2
min_alloc_log2: .quad 0
max_alloc_log2: .quad 0
```

The minimum and maximum allocation size are given in terms of logs of two, such that `9` is `512`, and `12` is `4096`. We input logs of two because a buddy tree must have a power of two nodes. Please note that the time to run one garbage collection pass increases as `max_alloc_log2 - min_alloc_log2` increases.

These two tunables are used to calculate the following variables:

```
.global min_alloc, max_alloc, bucket_count
min_alloc: .quad 0
max_alloc: .quad 0
bucket_count: .quad 0
```

Where `min_alloc` and `max_alloc` are the limits in bytes, and `bucket_count` is the maximum depth of the tree between the two log2 representations.

Using all this information, we allocate one mmap for one giant arena that starts with space for state tracking, and has the raw arena managed by the buddy allocator right after it.

```
.global buddy_mmap_ptr, base_ptr, base_end_ptr
buddy_mmap_ptr: .quad 0
base_ptr: .quad 0
base_end_ptr: .quad 0
```

We'll document the different other structures as we go along. There are two main structures in the buddy allocator: the buckets and the nodes.

16.2 Buddy Bucket Structure

The bucket structure is an array of intrusive doubly linked lists which are quick freelists for each allocation size. This array is `bucket_count` long and the largest bucket index represents the smallest allocation.

```
.global buckets_ptr
buckets_ptr: .quad 0
```

The "bucket number" of an allocation is the depth of the buddy tree where this allocation would go, and is used as a general identifier of size, even outside the bucket list.

16.2.1 bucket_for_request

Given a requested allocation size, round it up to the bucket that can hold that allocation.

TODO: This can actually be done without a loop with bsr.

```
.global bucket_for_request
bucket_for_request:
        shl     rdi, 3          # convert to bytes
        mov     rax, [bucket_count]
        dec     rax
        mov     rsi, [min_alloc]
bucket_for_request.loop:
        cmp     rsi, rdi        #
        jnb     bucket_for_request.done
        dec     rax
        shl     rsi, 1
        jmp     bucket_for_request.loop
bucket_for_request.done:
        ret
```

16.3 GC Headers

TODO: Figure out a better place for this to live.

Every node has a GC header. The tag bit on that header can be:

`0b0000 - 0x0 - Nat`
`0b0001 - 0x1 - Pin`
`0b0010 - 0x2 - Law`
`0b0011 - 0x3 - Clz (nf)`
`0b0100 - 0x4 - Clz (whnf)`
`0b0101 - 0x5 - Thunk`
`0b0110 - 0x6 - Megapin`

`0b0111 - 0x7 - Heap`
`0b1000 - 0x8 - Hash Table`
`0b1001 - 0x9 - Bump Slab`

16.4 Buddy Node Structure

The implementation of the buddy allocator used in the Linux kernel uses an optimized representation where each node is represented by a single bit, but we cannot use this representation because it is not traversable. For garbage collection purposes, we need to be able to traverse the buddy tree. We instead must represent three states:

```
.set    UNALLOCATED,    0
.set    INNER_NODE,     1
.set    ALLOC_GC,       2
.set    ALLOC_MANUAL,   3
```

This means that we need two bits to represent each node in the tree. We use a linearized binary tree, packing four items into each byte. We store this tree in the mmaped arena at `node_state_ptr`:

```
.global node_state_ptr
node_state_ptr: .quad 0
```

There are two ways to refer to a node:

- Directly by pointer to the first byte of a memory range.
- By raw index into the linearized node tree.

You can switch between these two representations as long as you have the "bucket number", the index which represents the depth of this allocation in the tree.

16.4.1 nodeget

Since nodes in our tree can be in one of three states, our lookup function must parse out those two bits from the byte.

```
.global nodeget
nodeget:                                    # nodeget(rdi=index):
        mov     ecx, edi                    #   copy index for shift
        and     ecx, 0x03                   #   ecx = index % 4
        shl     ecx, 1                      #   shift = (index % 4) * 2
        shr     rdi, 2                      #   byteIndex = index / 4
        mov     rax, [node_state_ptr]       #   load current base pointer
        movzx   eax, byte ptr [rax + rdi]   #   load byte
        shr     eax, cl                     #   offset byte
        and     eax, 0x03                   #   mask for answer
        ret                                 #   return
```

16.4.2 nodeset

The core primitives of our buddy tree must be thread safe, so we write them using atomics. Individual values in the tree are owned by different threads, but the packing means that setting one of the four values in a byte needs to not stomp other thread's work, so we use a `cmpxchg` loop to ensure we don't overwrite the value a different thread set.

```
.global nodeset
nodeset:                                    # nodeset(rdi=index, rsi=tristate):
        mov     ecx, edi                    #   copy index for shift
        and     ecx, 0x03                   #   ecx = index % 4
        shl     ecx, 1                      #   shift = (index % 4) * 2
        shr     edi, 2                      #   byteIndex = index / 4
        mov     r9d, 0x03                   #   r9d = 2 bit mask
        shl     r9d, cl                     #   shift mask into place
        and     esi, 0x03                   #   constrain to tristate
        shl     esi, cl                     #   shift state into place
        mov     rcx, [node_state_ptr]       #   load current base pointer
nodeset.try:                                # try:
        mov     al, [rcx + rdi]             #   load current full byte
        mov     r8b, al                     #   working copy
        and     r8b, r9b                    #   r8b = original & mask
        xor     r8b, al                     #   r8b = original with cleared bits
        or      r8b, sil                    #   r8b = new value inserted
        lock cmpxchg [rcx + rdi], r8b       #   CAS xchange al/[rcx+rdi]/r8b
        jnz     nodeset.try                 #   retry if contended
        ret                                 #   return
```

16.4.3 ptr_for_node

Given a node index and a bucket number, return the pointer to the actual node.

```
.global ptr_for_node
ptr_for_node:                   # ptr_for_node: (rdi=index, rsi=bucket)
        mov     rax, 1          #   base one
        mov     rcx, rsi        #   shl requires rcx
        shl     rax, rcx        #   rax = (1 << bucket)
        inc     rdi             #   + 1
        sub     rdi, rax        #   index = index - (1 << bucket) + 1

        mov     rcx, [max_alloc_log2]
        sub     rcx, rsi
        shl     rdi, rcx        # offset = index << (MAX_ALLOC_LOG2 - bucket)

        mov     rax, base_ptr
        add     rax, rdi        # rax = base_ptr + offset
        ret
```

16.4.4 node_for_ptr

Given a pointer and a bucket number, calculate the node index.

```
.global node_for_ptr
node_for_ptr:                   # node_for_ptr: (rdi=ptr, rsi=bucket)
        mov     rax, rdi
        sub     rax, base_ptr   # rax = ptr - base_ptr

        mov     rcx, [max_alloc_log2]
        sub     rcx, rsi
        shr     rax, rcx        # rax =>> MAX_ALLOC_LOG2 - bucket

        mov     rdx, 1
        mov     rcx, rsi
        shl     rdx, rcx        # rdx = 1 << bucket

        add     rax, rdx

        dec     rax
        ret
```

16.5 List Utilities

Buddy tree free lists are represented inline, where the memory for the free space linked list node lives right at where the allocation will be in the future. The free list is a doubly linked list for fast insertion and removal.

Each list starts with the following struct:

```
typedef struct list_t {
  struct list_t* prev;
  struct list_t* next;
} list_t;
```

16.5.1 list_init

The empty list points to itself.

```
.global list_init
list_init:                                  # list_init(rdi=list*):
        mov     [rdi], rdi                  #   list->prev = list;
        mov     [rdi+8], rdi                #   list->next = list;
        ret                                 #   return
```

16.5.2 list_push

```
.global list_push
list_push:                                  # list_push(rdi=list*,rsi=entry):
        ablptr  rdi                         #   input list pointer valid
        ablptr  rsi                         #   input new entry pointer valid
        mov     r8, [rdi]                   #   list_t* prev = list->prev;
        ablptr  r8                          #   input prev pointer valid
        mov     [rsi], r8                   #   entry->prev = prev;
        mov     [rsi + 8], rdi              #   entry->next = list;
        mov     [r8 + 8], rsi               #   prev->next = entry;
        mov     [rdi], rsi                  #   list->prev = entry;
        ret                                 #   return
```

16.5.3 list_remove

Unlinks a list entry from its list.

```
.global list_remove
list_remove:                                # list_remove(rdi=entry*):
        ablptr  rdi                         #   input entry pointer valid
        mov     r8, [rdi]                   #   list_t* prev = entry->prev;
        ablptr  r8                          #   prev pointer valid
        mov     r9, [rdi + 8]               #   list_t* next = entry->next;
        ablptr  r9                          #   next pointer valid
        mov     [r8 + 8], r9                #   prev->next = next;
        mov     [r9], r8                    #   next->prev = prev;
        ret                                 #   return
```

16.5.4 list_pop

```
.global list_pop
list_pop:                                   # list_pop(rdi=list*) -> rax:
        mov     r8, [rdi]                   #   list_t* back = list->prev;
        cmp     r8, rdi                     #   if back == list
        je      ret0                        #   then return null
        ablptr  r8                          #   assert pointer is in buddy heap
        mov     rdi, r8                     #   remove back from list
        call    list_remove                 #   list_remove(back)
        mov     rax, rdi                    #   set back as return value
        ret                                 #   return
```

16.6 Aligned Bit Trees

We perform liveliness marking against an aligned bit tree, which mirrors the structure of the buddy tree.

We have the logical structure where bit[0] is the head of a binary tree and its left child is `2*i + 1` and its right child is `2*i + 2`. The naive problem with this structure is that it is never word aligned. Meanwhile, we want to use this structure to be quadword aligned as much as possible so that we can do optimized walks over the data.

The compromise is that we just add one. Logical 0 is physical 1, logical 1 is physical 2, etc. This means all of level 0-5 sits in the first quadword, level 6 sits exactly in the second quadword, level 7 sits in the third and fourth quadword and so forth. All we have to do to achieve alignment is add a single bit.

16.6.1 aligned_bit_set

Atomically sets a specific bit, returning the previous value.

```
.global aligned_bit_set
aligned_bit_set:                            # aligned_bit_set(rdi=bit,rsi=v*):
        inc     rdi                         #   logical bit -> physical bit
        mov     rax, rdi                    #   rax = copy logical bit
        shr     rax, 6                      #   rax = byte_index
        and     rdi, 63                     #   rdi = bit_offset
        lock bts QWORD PTR [rsi+rax*8], rdi #   atomically set bit
        setc    al                          #   original value in CF to al
        movzx   eax, al                     #   al to return value
        ret                                 #   return
```

16.6.2 aligned_bit_get

```
.global aligned_bit_get
aligned_bit_get:                            # aligned_bit_get(rdi=bit,rsi=v*):
        inc     rdi                         #   logical bit -> physical bit
        mov     rax, rdi                    #   rax = copy logical bit
        shr     rax, 6                      #   rax = byte_index
        and     rdi, 63                     #   rdi = bit_offset
        bt      QWORD PTR [rsi+rax*8], rdi  #   test bit in byte
        setc    al                          #   original value in CF to al
        movzx   eax, al                     #   al to return value
        ret                                 #   return
```

16.7 Buddy Collection

While our buddy allocator provides explicit `buddy_malloc` and `buddy_free` calls for the cases where a process can control the lifecycle of data, since user pins are placed on the heap, the entire buddy allocator must provide a garbage collector. Our collection is concurrent: the collector runs on its own thread, and

16.7.1 Data Structures

When the concurrent garbage collector wants to collect, it pauses for every thread to mark its roots in the explore tree. Individual threads use `shallow_mark` to mark the bottom of this tree, and when everything is submitted, the sweep thread then uses `project_explore_tree` to make a binary bit tree used for exploration.

```
.global explore_tree_ptr
explore_tree_ptr: .quad 0
```

Because we don't want to perform a linear scan that of the entire explore tree, which will cause the kernel to back those pages, we instead keep track of the highest index in the final bucket layer so we can bound scanning.

```
.global explore_max_idx
explore_max_idx: .quad 0
```

Once the `explore_tree` is filled, it "explores", meaning it looks at completed explore structure to find live items and then recurse into its dependencies. The sweep tree should have every live allocation reachable from the items marked in the `explore_tree`.

```
.global sweep_tree_ptr
sweep_tree_ptr: .quad 0
```

But we need a second area to mark freshly malloced items that were generated while the exploration and sweeping were in progress. We don't want to overload the `sweep_tree_ptr` because you then confuse marking new items with whether you've recursively iterated through them, leading to undermarking.

```
.global new_allocation_ptr
new_allocation_ptr: .quad 0
```

These three bitmaps take a lot of space, and we don't want to have to zero them out, paging them into memory. They are very sparse, so we keep record the size of the bitmap in bytes so we can pass it to the kernel for clearing.

```
.global bitmap_bytes
bitmap_bytes: .quad 0
```

Finally, once we've filled the sweep tree, we perform a sweep where we move step by step through the tree. The actual state of the sweeper is visible as a global variable because sometimes the `buddy_free` process must interact and change the sweeping state if the two are accessing the same location. `sweep_level` will be -1 when not running.

```
.global sweep_level
sweep_level: .quad -1

.global sweep_node_ptr
sweep_node_ptr: .quad 0

.global sweep_action_ptr
sweep_action_ptr: .quad 0
```

16.7.2 Locks

Garbage collection of the buddy heap is concurrent, so we need some locks.

First is the `collector_lock`. A reader-biased reader/writer lock where malloc is a reader and the collector is a writer. This is the main lock that has to be taken whenever reading or writing to the `mark_state`.

The design is that the collector should only progress through the tree while there are no mallocs, because a malloc could be writing to the `mark_state` while calculating dependencies, which it is responsible for doing if the collector thread is sweeping.

```
.global collector_lock
collector_lock:
        .zero   8*7
```

`bucket_locks_ptr` will be allocated when as part of initialization to the right size of one u32 mutex for every bucket. The bucket locks are a per-bucket level lock needed to be taken before a thread reads or writes to the `buckets` or the `node_state` for that level.

When acquiring multiple locks, they should be acquired from the smallest allocation size (the highest bucket number) to the largest allocation size in the case of splitting.

```
.global bucket_locks_ptr
bucket_locks_ptr: .quad 0
```

The locking strategy is that you either take a write lock on `collector_lock`, or take a read lock on `collector_lock` and take a fine grained bucket lock. The sweeping process will take a coarse write `collector_lock`, while malloc will take a read `collector_lock` and will take fine grained locks for the buckets it needs to minimize contention inside malloc.

16.7.3 atomic_max

Given multiple threads recording indexes at the same time, we need a way to locklessly update a maximum number.

```
.global atomic_max
atomic_max:                                  # atomic_max(rdi=*, rsi=new_value):
        mov     rax, [rdi]                   #   load current value
atomic_max.retry:                            # retry:
        cmp     rsi, rax                     #   if new_value <= current
        jle     atomic_max.done              #   then don't change
        lock cmpxchg [rdi], rsi              #   if [rdi]==rax then [rdi]=rsi
        jne     atomic_max.retry             #   if ZF=0, exchange failed. retry.
atomic_max.done:                             # done:
        ret                                  #   return
```

16.7.4 project_or

Given a 64-bit word of child mark bits (packed as [child0, child1, child2, ... child63], where the children for parent i are at positions `2*i` and `2*i+1`), compute for each parent: parent's bit = `child_left` OR `child_right`.

This is done by:

- Shifting a copy of the 64-bit word
- ORing it with the original 64-bit word.
- Using PEXT to extract even-indexed bits (mask 0x5555...5555)

`project_or` expects its output to be pre-zeroed. It takes advantage of this fact by skipping running any bit manipulation code on any zero input since it'll result in a zero output.

```
.equ ODD_MASK, 0x5555555555555555
.global project_or
project_or:                                  # project_or(rdi=dst,rsi=src,rdx=l):
        push    r15                          #   save r15
        mov     r15, ODD_MASK                #   extraction mask
        xor     rcx, rcx                     #   i = 0
project_or.loop:                             # loop:
        mov     rax, [rsi+8*rcx]             #   load the 64-bit mark word
        test    rax, rax                     #   if mark word is zero
        jz      project_or.next              #   then skip the expensive pext
        mov     r8, rax                      #   copy the word
        shr     r8, 1                        #   shift right by 1 bit
        or      rax, r8                      #   combine pairs
        pext    r9, rax, r15                 #   extract every other bit
        mov     [rdi+4*rcx], r9d             #   store the result
project_or.next:                             # next:
        inc     rcx                          #   i++
        cmp     rcx, rdx                     #   if equals length
        jne     project_or.loop              #   then loop
        pop     r15                          #   restore r15
        ret                                  #   return
```

16.8 Buddy Sweeping

Sweeping is the second part of mark/sweep process for the buddy heap.

16.8.1 recursively_prune

Recursively walk down the tree starting from bucket `i` in `bucket` level, resetting the entire substructure. We handle the three states as such:

- `UNALLOCATED`: We just remove the item from the bucket list.
- `INNER_NODE`: We just recurse both ways.
- `ALLOC_GC`: This can be a no op. During pruning, we just abandon already allocated spans because they don't have any presence in the bucket list. But we may do various memory checks here
- `ALLOC_MANUAL`: It is always an error to encounter a manually allocated item while recursively pruning.

At the end of this, i, bucket and every child node of it will be UNALLOCATED in the `node_state` and will not appear in any freelists for allocation. (The caller is responsible for doing something with i, bucket.)

TODO: The assembly version of this has been temporarily removed and reverted to the C `recursively_prune_c` while we rework the collector to be concurrent, and add more checking code to the process.

17 The Persistence Heap

The runtime may provide a durable persistence heap which is written inside the current binary. This presents two XPLAN operations to the programmer: `precommit` and `commit`. While these two operations always are available, they only do something with the runtime has the persistence system compiled into it.

The primary point of the persistence system is to durably commit to a function to be run on process restart. As your XPLAN program runs, your XPLAN program can `commit` to a `function : Row Args -> ()` which will be run on the restart of the binary, which should resume the current state of the program. This is the core primitive that lets processes live forever from a committed checkpoint. The commit operation only writes data that hasn't been persisted.

The secondary point is to move durable data off of the buddy heap onto a heap where the Linux kernel can manage the paging of currently unneeded data. The `precommit` operation copies a tree of pins onto the persistence heap, returning the persisted root value. This doesn't change the committed startup function.

The purpose of `precommit` is to allow larger writes to happen in the background without blocking smaller commits which advance the actual state. The intended use-case here is that you persist states with a snapshot and an event log, and resume by loading the latest snapshot and then replaying all of the events since then. Writting a new event is fast, and will use `commit`. Snapshots are a lot bigger, and will use `precommit` on a background thread in order to commit a full state to disk without blocking the commit of additional events. Once a snapshot is fully written, it will be included in the next event, where each event references a snapshot whose state lags behind the event itself.

17.1 Low Level Page Storage

17.1.1 Mounting The Current Binary With Mmap

Normally, the Linux kernel will prevent a binary that's being executed from being opened for writing. If you try to open a writeable file descriptor, the kernel will return an `ETXTBSY` error. This means we have to go through some hoops that have to be performed right during a program's `_start` entry point.

The high level idea is that during `_start`, we create an anonymous file with `memfd_create`, copy the ELF portion of our binary to it, change the ELF entrypoint to `_restart` and then execveat this anonymous file.

The first thing we do is we open the current executable (`/proc/self/exe`) as read only, both so we can read the file, but also so we have a file descriptor to the current binary that we can use on the other side of the exec which is TOCTOU safe.

We then read current ELF header to figure out how much to copy, create a memfd, truncate it to the size of the ELF portion of the input file, copy the ELF data, change the ELF entrypoint to `_restart` which is the second stage execution, disable ASLR (since we need address space predictability for all of this to work), and then execveat the new anonymous program we just created.

```
.global boot_start
boot_start:
        ## Stack layout (32 bytes total):
        ##    [rbp - 8]   : exe file fd
        ##    [rbp - 16]  : memfd fd
        ##    [rbp - 24]  : ELF binary size
        ##    [rbp - 32]  : temporary buffer for patching binary
        mov     rbp, rsp
        add     rsp, -32

        mov     rax, SYS_OPEN
        lea     rdi, [proc_self_exe]    # open current executable
        mov     rsi, O_RDONLY           # no O_CLOEXEC so hold across exec
        syscall
        mov     [rbp - 8], rax          # store exe file fd; should always be 3

        mov     rdi, rax                # fd = exe file fd
        lea     rsi, [elf_header]       # buf
        mov     rdx, 64                 # count = 64
        mov     r10, 0                  # offset = null
        mov     rax, SYS_PREAD64        # Read ELF header into memory.
        syscall

        call    elf_binary_end
        mov     [rbp - 24], rax         # Set ELF binary size.

        lea     rdi, [plan_interpreter] # Anonymous memfd name
        mov     rsi, MFD_CLOEXEC        # Close the memffd on exec
        mov     rax, SYS_MEMFD_CREATE   # Create anonymous file descriptor
        syscall
        mov     [rbp - 16], rax         # store memfd fd

        mov     rdi, [rbp - 16]         # rdi=memfd fd
        mov     rsi, [rbp - 24]         # rsi=elf binary size
        mov     rax, SYS_FTRUNCATE      # Truncate memfd to ELF binary size
        syscall

        mov     rdi, [rbp - 16]         # out_fd = memfd
        mov     rsi, [rbp - 8]          # in_fd = exe file fd
        mov     rdx, 0                  # offset = 0
        mov     r10, [rbp - 24]         # count = total elf binary size
        mov     rax, SYS_SENDFILE       # Copy ELF exe portion to memfd
        syscall

        mov     rdi, [rbp - 16]         # out_fd = memfd
        lea     rsi, [_restart]         # address of new entry point
        mov     [rbp - 32], rsi         # write address to buffer
        lea     rsi, [rbp - 32]         # buf/rsi = buffer with _restart
        mov     rdx, 8                  # pointer is 8 bytes
        mov     r10, 24                 # e_entry is at 24 byte offset
        mov     rax, SYS_PWRITE64       # Write patched entry point to memfd
        syscall

        mov     rdi, ADDR_NO_RANDOMIZE
        mov     rax, SYS_PERSONALITY    # Disable ASLR for _restart
        syscall

        lea     rbx, [rsp + 32]         # rbx = original stack frame pointer
        mov     rax, [rbx]              # rax = argc

        mov     rdi, [rbp - 16]         # rdi = memfd fd
        lea     rsi, [empty_str]        # rsi = pointer to ""
        lea     rdx, [rbx + 8]          # rdx = argv
        lea     r10, [rbx+8+(rax+1)*8]  # r10 = envp
        mov     r8, AT_EMPTY_PATH       # r8 = flags
        mov     rax, SYS_EXECVEAT       # Restart program at _restart in memfd
        syscall
        ## If we return here, execveat() failed.

        add     rsp, 32                 # Restore the stack pointer.
        mov     rdi, 1                  # exit code 1
        call    syscall_exit_group      # and call exit
```

Once we call execveat, the kernel releases the execution lock on the original file, while still maintaining file descriptor 3 which points at it. Since Linux maintains symlinks to all open files in a process, and execveat doesn't close files without the `O_CLOEXEC` flag, we still have a read only file descriptor to the original file accessible with `/proc/self/fd/3`. Since there's no longer an execution lock on this file since we're now running from the memory fd, we can now open the file for writing.

As some book keeping, we need storage for our writable file descriptor:

```
.global self_fd
self_fd:        .quad 0
```

Then we just do normal process initialization before we call `main`.

```
_restart:
        lea     rax, boot_main
        mov     [chosen_main], rax

        lea     rdi, [proc_fd_path]     # rdi=/proc/self/fd/3
        mov     rsi, O_RDWR             # open for reading and writing
        mov     rax, SYS_OPEN           # Open the file for writing
        syscall
        mov     [self_fd], rax          # selfd = open(...)

        mov     rdi, rax                # fd = exe file fd
        lea     rsi, [elf_header]       # rsi = elf_header buffer
        mov     rdx, 64                 # count = 64
        mov     r10, 0                  # offset = null
        mov     rax, SYS_PREAD64        # Reread the ELF header.
        syscall

        jmp     sharedbegin
```

Finally, here's the helper function that parses the size of the ELF file. The definition is based off the of ELF headers, which we don't want to bundle into our project as a build dependency, and just calculates the length of the ELF portion of the binary:

```
.global elf_size_asm
elf_size_asm:
        movzx   eax, WORD PTR elf_header[rip+58]
        movzx   edx, WORD PTR elf_header[rip+60]
        imul    eax, edx
        add     eax, DWORD PTR elf_header[rip+40]
        ret
```

Finally, we have to store some read only constants used only in initialization:

```
.section .data
### ELF header of the executable loaded first thing on execution.
.align 16
elf_header:
        .zero   64

.section .rodata
proc_self_exe:
        .string "/proc/self/exe"
proc_fd_path:
        .string "/proc/self/fd/3"
plan_interpreter:
        .string "plan_interpreter"
empty_str:
        .ascii ""
```

This execution trampoline is only used in binaries that have the persistence system compiled in. It is not included in jar and the plan shell because it confuses gdb when you try to restart a binary. Those binaries use a small shim where _start just immediately calls _startmain.

17.1.2 Persistence Format

Our persistence heap starts at the first 4096 aligned page after the end of the ELF data. Our file format is two pages for superblocks, and then a series of data pages. We hardcode a page size of 4096 because that's the most common page size, and we don't want the file format to diverge on systems where that isn't true.

We allocate two pages, each dedicated to one superblock structure. Each superblock is on its own page since filesystems only guarantee that individual pages get committed. We then mmap the entire area of the file past this point.

```
const long page_size = 4096;
const long page_size_words = page_size / 8;

#define MAGIC_NUMBER 0x504E4c4e414c5000 // "PLANV1" in ASCII

#pragma pack(push, 1)       // Save alignment and set to 1 byte alignment
struct superblock {
  uint64_t magic;           // Magic number to identify our file type
  uint8_t sequence;         // Sequence number, wraps around 8-bit
  uint32_t val_checksum;    // Expected checksum for val
  Val val;                  // Tagged pointer into persistence heap.
  size_t write_offset;      // Current write position within the data area
  uint32_t checksum;        // Checksum of the superblock (excluding this field)
};
#pragma pack(pop)           // Restore previous alignment

struct superblock active_sb = {0};
off_t active_offset = 0;    // Binary location of current superblock
off_t inactive_offset = 0;  // Binary location of the other superblock
```

We have two superblocks with sequence numbers and CRC32 checksums, so that if we fail during writing of a new superblock, we can detect that case on startup and use the previous superblock.

We compute that by taking the checksum of the rest of the packed struct:

```
uint32_t calculate_superblock_checksum(struct superblock *sb) {
  return calculate_crc32c((const char*)sb,
                          sizeof(struct superblock) - sizeof(uint32_t));
}
```

17.1.3 Persistence System Initialization

Each main function has the choice to start the persistence subsystem or not. If a binary does want to initialize the system, they call init_persistence with the file descriptor of the file they want to mount. This file descriptor is usually the self_fd descriptor that was opened during _restart, but can also be a target file when building an image for the first time.

init_persistence will calculate the locations of the superblocks, mount the region past the superblocks in an mmap, figure out which is the active superblock (if any), and record the bounds of the current persistence image:

```
void init_persistence(int fd) {
  persistence_fd = fd;

  struct stat st;
  if (syscall_fstat(persistence_fd, &st) < 0) {
    syscall_close(persistence_fd);
    die("Failed to get file size");
  }
  filesize = st.st_size;

  off_t elf_size = elf_size_asm();
  off_t sb1_offset = round_up_align(elf_size, page_size);
  off_t sb2_offset = sb1_offset + page_size;

  // Data starts at the third page
  size_t data_start = sb1_offset + 2 * page_size;
  headersize = data_start;

  // Mount the data into memory. We do this before processing the superblocks
  // so we can do a top level tree validation to make sure it's as written
  // correctly.
  //
  // TODO: Lay out memory maps so they are all address space, collaborating
  // with the buddy allocator.
  const int map_prot = PROT_READ | PROT_WRITE;
  const int map_flags = MAP_SHARED | MAP_FIXED;
  const u64 base_size = (1UL << 39);
  persistence_start = syscall_mmap((void*)base_addr,
                                   base_size, rw,
                                   map_flags, persistence_fd,
                                   data_start);

  // Read and check the validity of the two superblocks.
  struct superblock sb1, sb2;
  bool have_sb1 = false, have_sb2 = false;
  if (filesize >= sb1_offset + sizeof(struct superblock)) {
    have_sb1 = read_superblock(persistence_fd, sb1_offset, &sb1);
  }
  if (filesize >= sb2_offset + sizeof(struct superblock)) {
    have_sb2 = read_superblock(persistence_fd, sb2_offset, &sb2);
  }

  find_active_superblock(sb1, have_sb1, sb1_offset,
                         sb2, have_sb2, sb2_offset);

  persistence_cur = persistence_start + active_sb.write_offset;
  persistence_end = persistence_start + (base_size / 8);
}
```

17.1.4 Reading a Superblock

When we read or write a superblock, we do it by reading directly from the file, instead of reading or writing through a memory mapped page. (We are following what LMDB does here.)

We verify that we can read the right amount of data, verify it has the right magic number, verify that the checksum of the superblock is right, and verify that the checksums for the value the superblock points to is correct.

Returns true if everything is valid, false if anything is invalid.

```
bool read_superblock(int fd, off_t offset, struct superblock *sb) {
  ssize_t bytes_read = syscall_pread64(
      fd, (char*)sb, sizeof(struct superblock), offset);

  if (bytes_read != sizeof(struct superblock)) {
    if (bytes_read < 0 || (bytes_read == -22 /* EINVAL */)) {
      // This might be a new file or truncated file
      return false;
    }
    die("Failed to read superblock");
  }

  if (sb->magic != MAGIC_NUMBER) {
    printf("Invalid superblock (wrong magic number) at offset %ld\n",
           (long)offset);
    return false;  // Block is not valid
  }

  uint32_t expected_checksum = sb->checksum;
  uint32_t actual_checksum = calculate_superblock_checksum(sb);
  if (expected_checksum != actual_checksum) {
    printf("Superblock corruption detected (checksum mismatch) at offset %ld\n",
           (long)offset);
    return false;  // Block is not valid
  }

  expected_checksum = sb->val_checksum;
  actual_checksum = crc32_checksum_for(sb->val);
  if (expected_checksum != actual_checksum) {
    printf("Superblock corruption detected (content mismatch) at offset %ld\n",
           (long)offset);
    return false;  // Block is not valid
  }

  return true;
}
```

17.1.5 Writing a Superblock

Writing a superblock calculates and sets the checksum of the rest of the block, and then writes and fsyncs the file descriptor to make sure the data is persisted.

```
void write_superblock(int fd, off_t offset, struct superblock *sb) {
    sb->checksum = calculate_superblock_checksum(sb);

    ssize_t bytes_written = syscall_pwrite64(
        fd, (char*)sb, sizeof(struct superblock), offset);
    if (bytes_written != sizeof(struct superblock)) {
      die("Failed to write superblock");
    }

    if (syscall_fsync(fd) == -1) {
      die("Failed to fsync superblock");
    }

    // Swap offsets.
    off_t tmp = active_offset;
    active_offset = inactive_offset;
    inactive_offset = tmp;
}
```

17.1.6 Determining Which Superblock To Use

On startup, in init_persistence, we have to initialize the state of which superblock is active and which one isn't. We look at which superblocks exist in the file, and initialize the state to the newest valid superblock, initializing a new superblock if there's no persistence data in the file.

```
void find_active_superblock(struct superblock sb1,
                            bool have_sb1,
                            off_t sb1_offset,
                            struct superblock sb2,
                            bool have_sb2,
                            off_t sb2_offset) {
  if (have_sb1 && have_sb2) {
    // Handle wraparound by checking if the difference (considering
    // unsigned overflow) is less than 128
    if ((uint8_t)(sb1.sequence - sb2.sequence) < 128) {
      active_sb = sb1;
      active_offset = sb1_offset;
      inactive_offset = sb2_offset;
    } else {
      active_sb = sb2;
      active_offset = sb2_offset;
      inactive_offset = sb1_offset;
    }
  } else if (have_sb1) {
    active_sb = sb1;
    active_offset = sb1_offset;
    inactive_offset = sb2_offset;
  } else if (have_sb2) {
    active_sb = sb2;
    active_offset = sb2_offset;
    inactive_offset = sb1_offset;
  } else {
    // Write the initial superblock
    active_sb.magic = MAGIC_NUMBER;
    active_sb.sequence = 1;
    active_sb.val_checksum = natural_crc32(0);
    active_sb.val = 0;
    active_sb.write_offset = 0;
    write_superblock(persistence_fd, sb1_offset, &active_sb);

    active_offset = sb1_offset;
    inactive_offset = sb2_offset;
  }
}
```

17.1.7 Allocating Raw Pages

The low level primitive in the persistence file is the allocation of pages. Right now, we allocate a number of pages at the end of the file. This function only makes sure the file is large enough to contain the requested allocation (and calling ftruncate if it isn't) and

This function does no syncing. It just hands out a pointer to persistence backed memory.

```
u64* allocate_persistence_space_for(size_t sz) {
  size_t required_size = headersize
                       + ((persistence_cur - persistence_start) * 8)
                       + (sz * 8);

  if (filesize < required_size) {
    size_t new_size = round_up_align(required_size, page_size);
    if (syscall_ftruncate(persistence_fd, new_size) == -1) {
      die("Failed to extend file");
    }

    filesize = new_size;
  }

  // Our current pointer is always page aligned.
  u64* now = persistence_cur;
  persistence_cur += round_up_align(sz, page_size_words);
  return now;
}
```

17.1.8 Msyncing pages

The caller to allocate_persistence_space_for is responsible for msyncing the pages once they are written to.

```

#define MS_SYNC 4

void msync_region(u64* begin, u64 word_size) {
  // Round word_size turned into bytes to the nearest page.
  u64 rounded_size = round_up_align(word_size * 8, page_size);

  // Sync the region
  if (syscall_msync(begin, rounded_size, MS_SYNC) == -1) {
    die("Failed to msync data");
  }
}
```

17.2 Persisting PLAN values

17.3 Persistence Collection

17.4 Scrubing and Validation

All pins persisted have the crc32 bit set, meaning they have checksums to make sure they point to the right data and haven't been corrupted either by the system or by the garbage collector.

We calculate the checksums with the x86 crc32 instruction, which does Castagnoli CRC.

17.4.1 Calculating CRC32 Checksums

For natural numbers, we just do a single CRC instruction on the direct natural number.

```
u64 natural_crc32(u64 nat) {
  __int64_t crc = 0xFFFFFFFF;
  crc = _mm_crc32_u64(crc, nat);
  crc ^= 0xFFFFFFFF;
  return crc;
}
```

For general PLAN values, note that we run the CRC32 loop over the heap header one space before the pointer to the item:

```
u64 crc32_checksum_for(Val item) {
  if ((item >> 63) == 0) {
    return natural_crc32(item);
  } else {
    u64* ptr = PTR(item);
    u64 bitsz = get_bitsz(item);
    u64 wordsz = (bitsz + 63) / 64;

    return calculate_crc32c((char*)(ptr - 1), (wordsz + 1) * 8);
  }
}
```

All usages of CRC32 are bound at load time by the following implementation, which is unrolled to operate in qwords for efficiency.

```
.global calculate_crc32c
calculate_crc32c:                                   # crc32c(rdi=data, rsi=bytesz):
        mov     eax, -1                             #   crc initialized to all ones
        test    rsi, rsi                            #   if size == 0
        jz      ret0                                #   then return zero
        mov     rcx, rsi                            #   rcx = len
        shr     rcx, 3                              #   blocks = len / 8
        and     rsi, 7                              #   rsi = len % 8 (tail size)
        test    rcx, rcx                            #   if blocks == 0
        jz      calculate_crc32c.tail4              #   then jump past loop8
calculate_crc32c.loop8:                             # loop8:
        crc32   rax, qword ptr [rdi]                #   process 8 bytes
        add     rdi, 8                              #   src += 8
        dec     rcx                                 #   blocks--
        jnz     calculate_crc32c.loop8              #   if remaining blocks, loop
calculate_crc32c.tail4:                             # tail4:
        test    rsi, 4                              #   if len & 4 == 0
        jz      calculate_crc32c.tail2              #   then goto tail2
        crc32   eax, dword ptr [rdi]                #   process 4 bytes
        add     rdi, 4                              #   src += 4
calculate_crc32c.tail2:                             # tail2:
        test    rsi, 2                              #   if len & 2 == 0
        jz      calculate_crc32c.tail1              #   then goto tail1
        crc32   eax, word ptr [rdi]                 #   process 2 bytes
        add     rdi, 2                              #   src += 2
calculate_crc32c.tail1:                             # tail1:
        test    rsi, 1                              #   if len & 1 == 0
        jz      calculate_crc32c.done               #   then goto done
        crc32   eax, byte ptr [rdi]                 #   process 1 byte
calculate_crc32c.done:                              # done:
        not     eax                                 #   eax = eax xor 0xffffffff
        ret
```

17.4.2 Validating Each Item

For each item, we check that its checksum matches or we kill the process. If the item is a tagged pointer, we also validate that the embedded crc is unchanged.

```
void validate_item(Val item, u64 expected) {
  __int64_t crc = crc32_checksum_for(item);

  if ((item >> 63) == 1) {
    u64* ptr = PTR(item);
    if (ptr[-2] != crc) {
      printf("Item pointer CRC32 differs for %lx: ptr[-2]=%lx, actual=%lx\n",
             item, ptr[-2], crc);
      die("scrub failed");
    }
  }

  if (expected != crc) {
    printf("CRC32 differs for %lx: expected=%lx, actual=%lx\n",
           item, expected, crc);
    die("scrub failed");
  }
}
```

17.4.3 Scrubbing An Entire Pin Tree

Our superblock structure records both a PLAN value and the expected crc32 of the value. So to scrub a whole recursive pin structure, we set up an empty red-black tree with a bump allocator so that we can track the regions of memory we've already checked, and we then start the recursive process:

```
void scrub(Val item, u64 expected) {
  bump_alloc_t alloc;
  bump_alloc_init(&alloc, sizeof(rb_node));

  rb_tree tree;
  rb_init(&tree);

  scrub_item(item, expected, &tree, &alloc);

  bump_alloc_free(&alloc);
}
```

scrub_item is the recursive step, where we check each item in a tree of pins. It uses the same mark_item_in_tree tracking system used by the persistence collector, because pin dependencies are massively duplicated in practice.

This structure checks the validity of a single item, by checking that its own crc32 that proceeds it matches the expected checksum, that the item's bytes match the checksum, and then recurring based on whether this is a pin or a megapin. This leverages megapin structure so that we don't have to recur through the pin tree and can just linearly walk the megapin index.

```
void scrub_item(Val item, u64 expected, rb_tree* tree, bump_alloc_t* a) {
  if ((item >> 63) == 0) {
    // Check a direct reference.
    u64 crc = natural_crc32(item);
    if (crc != expected) {
      printf("Header hash differs from expected for %lx\n", item);
      die("scrub failed");
    }

    return;
  }

  if (mark_item_in_tree(item, tree, a)) {
    return;
  }

  u64* ptr = PTR(item);
  if (ptr[-2] != expected) {
    printf("Header hash differs from expected for %lx\n", item);
    die("scrub failed");
  }

  validate_item(item, expected);

  u64 pin_count = ptr[2];
  if (ismegapin(item)) {
    u64 megapin_count = ptr[6 + 2 * pin_count];

    u32* pin_crcs = (u32*)(ptr + 6 + 2 * pin_count + 1 + 2 * megapin_count);

    // nonrecursively scrub the pins
    for (size_t i = 0; i < pin_count; ++i) {
      if (mark_item_in_tree(ptr[6 + i], tree, a))
        continue;

      validate_item(ptr[6 + i], pin_crcs[i]);
    }

    // recursively scrub the megapins.
```

```
    for (size_t i = 0; i < megapin_count; ++i) {
      scrub_item(ptr[6 + 2 * pin_count + 1 + i],
                 pin_crcs[pin_count + i],
                 tree, a);
    }
  } else {
    u32* pin_crcs = (u32*)(ptr + 6 + pin_count);
    for (size_t i = 0; i < pin_count; ++i) {
      scrub_item(ptr[6 + i], pin_crcs[i], tree, a);
    }
  }
}
```

18 Naive Compiler

The native runtime will include an extremely simple. See `judge.pdf` for the details of that proposal.

However, in the meantime, we still need to recognize primop wrappers, because otherwise primops have a ton of overhead.

18.1 Wrappers

A primop wrapper is a function which take a certain number of arguments, and then directly calls a single primop with them. These are used everywhere, since the primop ABI is uncurried, but we want currying because it's faster and cleaner.

Here's a simple example, a law which invokes the Add operation:

```
(a b)&(<15> (%Add a b)) =>

    {0 2 (0 <15> ((0 "Add" 1) 2)}
```

18.1.1 Recognizing Wrappers

In order to recognize a wrapper, we need a routine which recognizes this pattern, and tells use the op, arity, and hd. Or just tells us that it didn't match.

If it is a wrapper, we can then look that up and see if it is a known operation, and then replace the pin/law executioner to directly call that operation.

These laws all have a simple shape:

```
{_ arity body(arity)}

body(arity) = (0 <op:Nat> adt(arity))

adt(0) = hd:Nat (where n>arity)
adt(0) = [hd:Nat]
adt(n) = (0 adt(n-1) n)
```

18.1.2 match

```
ARG rsi -- The law to inspect
RET rax -- The primop method
RET rdx -- The primop class
RET r8  -- The law arity (zero if no match)
```

```
match:                                     # match: rdi/law
        arity_  r8, rdi                    #   r8  = law.arity
        body_   rax, rdi                   #   rax = law.body
        jnkal   rax, match.none            #   if (body !~ 0[a b]) goto none
        ix0_    rdx, rax                   #   rdx = body.ix(0)
        jnpin   rdx, match.none            #   if (rdx !~ <i>) goto none
        unpin   rdx                        #   rdx = rdx.item
        jheap   rdx, match.none            #   if !direct(rdx) goto none
        ix1     rax                        #   x = body.ix(1)
        mov     rcx, r8                    #   i = arity
match.loop:                                # loop: rdx/op r8/args rax/x rcx/i
        test    rcx, rcx                   #   if (i == 0)
        jz      match.head                 #     head
        jnkal   rax, match.none            #   if !(x ~ 0[a b]) goto none
        ix1_    r9, rax                    #   r9=b
        cmp     r9, rcx                    #   if (b != i)
        jne     match.none                 #     goto none
        ix0     rax                        #   x=a
        dec     rcx                        #   i--
        jmp     match.loop                 #   goto loop
match.head:                                # head: rdx/op r8/arity rax/head
        jdirect rax, match.noquote         #   if direct(x) goto noquote
        jnquo   rax, match.none            #   if !(x ~ 0[k]) goto none
        dref    rax                        #   x = k
match.found:                               # found:
        ret                                #   return
match.noquote:                             # noquote: rdx/op r8/arity rax/hd
        cmp     rax, r8                    #   if (x > arity)
        ja      match.found                #     then goto found
match.none:                                # none:
        xor     r8d, r8d                   #   arity=0
        xor     eax, eax                   #   key=0
        xor     edx, edx                   #   op=0
        ret                                #   return
```

19 Profiling

19.1 Profile Format

The native runtime has built in lightweight profiling events. A thread can use an XPLAN operation to start and stop recording profiling events, and another to return the raw data to the caller.

Each profiling record is two values, where the first is a 63-bit direct number with three bits of event tag information and 60-bits of unix time at 8ns resolution. The second is event type specific, but for begin/end events, is the actual law value being run. The trick here is that the native runtime is trying to lay out memory directly in a form that is a) fast to produce, b) directly handable to the programs.

That memory is laid out as a linked list of closures, where the most recent page is at the start of the list. Therefore, the format for each profiling page is:

```
5[prev_page a1 a2 b1 b2 c1 c2 ...]
```

This format moves most complexity of profiling from the native runtime to the running program.

The only global variables needed are a pointer to the current allocation, a pointer to where in that allocation to write to next, and an end pointer to track when we need to overflow and allocate a new page.

```
.section .data
.global profile_base, profile_next, profile_end
profile_base:   .quad 0
profile_next:   .quad 0
profile_end:    .quad 0
```

19.2 Time Format

We choose a 60-bit 8ns resolution because that's good up until the year 2262 while being close to a resolution that's drowned out by kernel overhead.

We use the following macro to turn a timespec pointer filled by the kernel into our time format:

```
.macro PTIME60 dst, src
        mov     \dst, [\src+0]             #   rax = tv_sec (time_t)
        mov     r12, [\src+8]              #   r12 = tv_nsec (long)
        imul    \dst, \dst, 125000000      #   ticks = tv_sec * (1e9 / 8)
        shr     r12, 3                     #   tv_nsec >>= 3
        add     \dst, r12                  #   sum ticks
        mov     r12, 60                    #   literal 60
        bzhi    \dst, \dst, r12            #   mask 60 bits.
.endm
```

19.3 Push Profile Page

Since the profiling results are a linked list of pages, pushing a profile page is both the main initialization function and what gets called repeatedly to add a new item to the linked list.

```
.set    PROFILE_SIZE,   4088
push_profile:                              # push_profile:
        mov     rdi, PROFILE_SIZE          #   rdi = PROFILE_SIZE
        mov     rsi, 0x004                 #   rsi = closure tag
        call    buddy_malloc               #   buddy_malloc(PROFILE_SIZE, 4)
        test    rax, rax                   #   if malloc returned null
        js      push_profile.oom           #   then handle oom
push_profile.prepareclosure:               # prepareclosure:
        mov     r8, [profile_base]         #   check current profile_base
        test    r8, r8                     #   if profile_base == NULL
        jz      push_profile.write         #   then skip cleanup
        push    rax                        #   save rax
        call    zero_remaining_profile     #   zero out remaining profile page
        pop     rax                        #   restore rax
        mov     r9, 0xD050000000000000     #   write correct header
        or      r8, r9                     #   or header with profile_base ptr
push_profile.write:                        # write
        mov     qword ptr [rax], 5         #   Set hd to 5
        mov     [rax + 8], r8              #   5[profile_base a1 a2 b1 b2...]
        mov     [profile_base], rax        #   save new profile_base
        mov     rcx, rax                   #   end = start
        add     rcx, PROFILE_SIZE*8        #   end = start + size
        mov     [profile_end], rcx         #   set end
        add     rax, 16                    #   increment to first entry
        mov     [profile_next], rax        #   set next write location
        ret
push_profile.oom:
        ud2
```

19.4 Zero Remaining Profile Page

We don't want to initialize a full profile linked list closure at allocation time because that will destroy cache performance. We only zero the remaining unused structure when we have to either push another profile page to the reverse linked list structure, or when we have to hand this chain of pages to the program.

```
.global zero_remaining_profile
zero_remaining_profile:                    # zero_remaining_profile:
        mov     rdi, [profile_next]        #   rdi = profile_next
        mov     rcx, [profile_end]         #   rcx = profile_end
        sub     rcx, rdi                   #   rcx = remaining space
        shr     rcx, 3                     #   bytes to words
        xor     eax, eax                   #   store 0
        rep     stosq                      #   memset
        ret                                #   return
```

19.5 recordevent

This is the low level recording implementation for everything else. This takes an input of the tag and the

2nd value to write to the log.

```
### TODO: This calls the SYS_clock_gettime call. YOU REALLY DO NOT WANT TO DO
### THIS IN PRODUCTION CODE. This introduces SYSCALL OVERHEAD to a common
### action we're doing all the time. The correct thing to do is to call the
### VDSO version of this, but that's going to require parsing the auxiliary
### vector at startup, then do minimal ELF parsing to find lookup the pointer
### to `__vdso_clock_gettime`. That's all hard. So for getting from 0 to 1, we
### punt and do the wrong thing. But the data is going to be quasi-invalid
### until we do that.
.global recordevent
recordevent:                                # recordevent(rdi=tag, rsi=2nd val):
        sub     rsp, 32                     #   stack space for timespec*
        mov     [rsp + 16], rdi             #   save tag
        mov     [rsp + 24], rsi             #   save 2nd val
recordevent.check:                          # check free space:
        mov     rax, [profile_next]         #   get current next pointer
        add     rax, 16                     #   increment by 2 64bit words.
        mov     rcx, [profile_end]          #   get end pointer
        cmp     rax, rcx                    #   if (next >= profile_end)
        jae     recordevent.overflow        #   then handle overflow
recordevent.askfortime:                     # ask kernel for time:
        mov     rax, 228                    #   SYS_clock_gettime
        mov     rdi, 1                      #   CLOCK_MONOTONIC
        mov     rsi, rsp                    #   rsi = &timespec
        syscall                             #   call kernel
recordevent.record:                         # record record:
        PTIME60 rax, rsp                    #   timespec to 60-bit 8ns time.
        shl     rax, 3                      #   make room for tag
        mov     rdi, [rsp + 16]             #   rdi = input tag
        or      rax, rdi                    #   OR time with tag
        mov     rcx, [profile_next]         #   get current pointer
        mov     [rcx], rax                  #   low word: time+type tag
        mov     rdi, [rsp + 24]             #   retrieve 2nd val
        mov     [rcx + 8], rdi              #   high word: law pointer
        lea     rcx, [rcx + 16]             #   advance next pointer by 16
        mov     [profile_next], rcx         #   save updated pointer
        add     rsp, 32                     #   restore stack frame
        ret                                 #
recordevent.overflow:                       # overflow:
        ppush   rdi, rsi                    #   save input during allocation
        call    push_profile                #   new profile frame
        ppop    rdi, rsi                    #   restore input
        jmp     recordevent.askfortime      #   continue in before path
```

19.6 opprofile

The main generator of profiling events is the opprofile function, which is the jet of `Profile` function. The `Profile` function is defined as just returning its second argument.

In the fast path, when profiling is disabled (ie there are no profiling frames to write to), the overhead is just one fused compare-and-jump and one register move.

We must absolutely prevent references to GC1 from leaking into GC2 because that breaks our garbage collection semantics. So we handle this by ignoring any `Profile` call that contains data on a thread's Cheney heap. All long term Law objects should be on GC2 or GC3.

```
opprofile:                                  # opprofile(rdi=law rsi=ret val):
        cmp     qword ptr [profile_next], 0 #   if profiling is enabled
        jne     opprofile.start             #   then jump to profile handling
opprofile.default:                          # default:
        mov     rax, rsi                    #   setup 2nd argument
        JRAX                                #   evaluate 2nd argument
opprofile.start:                            # start:
        jdirect rdi, opprofile.savespace    #   if direct, start immediately
        ptr_    rcx, rdi                    #   rcx = PTR(rdi)
        cmp     rcx, r14                    #   if (1st < heap_end)
        jb      opprofile.checkincheney     #   then check if in cheney heap
opprofile.savespace:                        # save space:
        sub     rsp, 16                     #   stack space for two
        mov     [rsp], rdi                  #   save law pointer
        mov     [rsp + 8], rsi              #   save eval thunk
opprofile.recordbegin:                      # record begin:
        mov     rsi, rdi                    #   rsi = "2nd val"
        mov     rdi, 0                      #   rdi = type tag 0 (begin)
        call    recordevent                 #   recordevent(0, law ptr)
opprofile.eval:                             # eval:
        mov     rsi, [rsp + 8]              #   get saved thunk
        EVAL    rsi                         #   evaluate it
        mov     [rsp + 8], rsi              #   put saved evaluation
opprofile.recordend:                        # record end:
        mov     rsi, [rsp]                  #   rsi = "2nd val"
        mov     rdi, 1                      #   rdi = type tag 1 (end)
        call    recordevent                 #   recordevent(0, law ptr)
opprofile.cleanup:                          # cleanup:
        mov     rax, [rsp + 8]              #   retrieve result
        add     rsp, 16                     #   undo stack space
        JRAX                                #   evaluate 2nd argument
opprofile.checkincheney:                    # check in cheney:
        cmp     rcx, [heap_addr]            #   if (PTR(rdi) >= heap_addr)
        jae     opprofile.default           #   then can't record value
        jmp     opprofile.savespace         #   otherwise continue recording
```

19.7 SetProfEnabledOp

The profiling system is explicitly enabled and disabled by the using code. This XPLAN operation takes 0 or 1 for whether it's enabled.

```
SetProfEnabledOp:                           # SetProfEnabledOp(rdi=enabled):
        test    rdi, rdi                    #   if (rdi == 0)
        jz      SetProfEnabledOp.disable    #   then disable requested
SetProfEnabledOp.enable:                    # enable:
        cmp     qword ptr [profile_next], 0 #   if profiling already enabled
        jne     ret0                        #   then just return 0
        call    push_profile                #   push a profile frame to start
        ret                                 #   done
SetProfEnabledOp.disable:                   # disable:
        mov     qword ptr [profile_base], 0 #   profile_base = null
        mov     qword ptr [profile_next], 0 #   profile_next = null
        mov     qword ptr [profile_end], 0  #   profile_end = null
        ret                                 #   let gc collect dead frames
```

19.8 GetProfOp

The calling program must periodically call this XPLAN operation to return the current profiling samples.

```
GetProfOp:                                  # GetProfOp(rdi=()):
        cmp     qword ptr [profile_next], 0 #   if profiling is enabled
        jne     GetProfOp.enabled           #   then jump to profile handling
        mov     rax, 0                      #   else return 0
        ret                                 #   return
GetProfOp.enabled:                          # enabled:
        call    zero_remaining_profile      #   zero the remaining profile
        mov     rax, [profile_base]         #   get current closure base ptr.
        mov     r8, 0xD050000000000000      #   write correct header
        or      rax, r8                     #   make rax tagged pointer
        mov     qword ptr[profile_base], 0  #   reset profile_base stack
        push    rax                         #   save result
        call    push_profile                #   push a new profile buffer
        pop     rax                         #   restore result
        ret                                 #   return
```

20 Seed

20.1 The Boot Seed

The first argument to the `plan` program is a seed to run which contains an XPLAN program.

xseed programs have a calling convention similar to C programs: The seed file is loaded, the function is passed the rest of the command-line arguments as an array of nats, the result is evaluated, cast to a nat, and then passed to the `exit(2)` syscall.

This section will document the seed format, and the implementation of the seed loader.

20.2 The Seed Format

20.2.1 Introduction

This will attempt to communicate the basic idea of Seed without getting caught up in implementation concerns. The idea is that, if you understand the basic format layout, that should give you pretty strong intuitions about the how encoding and decoding work.

I will avoid talk about /why/ choices were made, since many choices are informed by implementation concerns, and I want to avoid getting caught up in that. Instead, I will only describe the format itself.

A seed file implements a template for a PLAN value, with a special number of parameters. These parameters will need to be passed into the loader routine.

Because PLAN values often contain a lot of shared structure, this is represented explicitly in the format. The format contains a number of fragments which can refer to each-other, and the final fragment is the actual resulting value.

20.2.2 The Header

The first 40 bytes of a Seed is a header which contains 5 64-bit words. All words in the format are represented in least-significant bit order.

- The number of parameters for the template.
- The number of multi-word numbers.
- The number of 64-bit numbers.
- The number of 8-bit numbers.
- The number of fragments.

20.2.3 Numbers

After the header, the format contains all of the numbers, in descending order.

- The first thing is the width of each multi-word number.
- And then the actual data for each multi-word number
- And then the data for each single-word number.
- And then the data for each single-byte number.

20.2.4 Scopes and References

At each point in the seed, everything that came before is "in scope". All of the arguments, all of the numbers, and every preceding tree binding.

For a certain number of bindings in scope, we require a certain number of bits. For example, if there are four things in scope, each reference will require two bits.

20.2.5 Fragments

Trees are encoded as a bit-packed recursive structure. Each node is a reference applied to zero or more parameters. This is encoded as:

- The size of the size in unary.
- The size in binary.
- The actual reference bits
- Zero or more following nodes, one per parameter.

Because the high-bit of the actual size is always 1, so the high bit is omitted in order to save space. The only edge-case is the case where there are zero parameters, but this is easy to handle.

Here are some examples of sizes and how they are encoded. Something tricky here is that these examples are listed LSB-first, which means that the actual binary values need to be read in reverse.

```
0 -> 1.
1 -> 01.
2 -> 001.0
3 -> 001.1
4 -> 00010.00
5 -> 00010.11
6 -> 00010.01
7 -> 00010.11
8 -> 00011.000
```

20.2.6 Encoding Pins and Laws

The format only includes numbers and applications, so there actually isn't any way to encode pins and laws.

However, the resulting template does not need to be in normal form, so we can just take MkPin and MkLaw as arguments. And we actually do not need to take MkLaw as an argument, because it can be constructed using just MkPin.

The specific conventions around which argument are passed in depends on the context.

When loading a boot seed, we pass in MkPin and we construct MkLaw ourselves. Since the calling convention for MkLaw is going to get more complicated, we likely should pass in MkLaw as well.

When loading a single node of a Pin DAG, we currently pass in MkPin but only use it to construct laws, since all of the sub-pins are passed in as references.

Since the calling convention for MkLaw is going to get more complicated, we likely should pass in both MkPin and MkLaw during boot, and only MkLaw when working with individual DAG nodes.

20.2.7 Important Properties

Seed has a number of properties which make it a good format for our uses:

- Decoding is fast, and requires very little code.
- Decoding requires no significant data structures, just an array.
- Loading numbers is a trivial memory read, as they are stored in aligned memory, in a machine-native format.

20.3 The Seed Loader

The built-in loaded is designed to be as small as possible, and does no input validation.

As a result, it is not safe. This is designed only for use with known seeds during bootstrap. If given an invalid seed, this will segfault.

There is no additional security concern here, since this is the first thing that the process does, and we are already reading this as an XPLAN program and running it, which can perform arbitrary system calls.

20.3.1 mmapfile

Given a path and a pointer to a statbuf (structure produced by fstat syscall), this opens a file, gets the size, and then loads the entire file into memory using mmap.

Returns the buffer, uses SysV calling conventions.

```
.global mmapfile
mmapfile:                                   # rdi=path rsi=statbuf
        push    r12
        push    rbx
        mov     r12, rsi                    # r12=statbuf
        xor     esi, esi                    # flags=0
        xor     edx, edx                    # mode=0
        call    syscall_open_chk            # open(path, flags, mode)
        mov     rbx, rax                    # fd to global
        mov     rdi, rax                    # fd
        mov     rsi, r12                    # statbuf = &seedstat
        call    syscall_fstat_chk           # fstat(fd, statbuf)
        xor     edi, edi                    # addr=0
        mov     rsi, [r12+48]               # len=file.size
        mov     edx, 1                      # prot=1
        mov     ecx, 2                      # flags=2
        mov     r8, rbx                     # fd = *seedfd;
        xor     r9d, r9d                    # off=0
        call    syscall_mmap_chk            # mmap(ptr,len,prot,flags,fd,off)
        mov     r12, rax                    # save buffer
        mov     rdi, rbx                    # fd = *seedfd
        call    syscall_close_chk           # close(rd)
        mov     rax, r12                    # restore result=buffer
        pop     rbx
        pop     r12
        ret                                 # return
```

20.3.2 seedfile

Given a filename, load and return the seed, following SysV calling conventions.

```
.global seedfile # RPN testing system calls this.
seedfile:
        sub     rsp, 160                 # 144+padding
        mov     rsi, rsp                 # rsi = &stat (stack local)
        call    mmapfile                 # rax = ptr
        mov     [rsp+152], rax           # save buf
        mov     rdi, rax                 # buf
        call    seed                     # seed(buf)
        mov     rdi, [rsp+152]           # ptr = buf
        mov     rsi, [rsp+48]            # len = stat.st_size
        mov     [rsp+48], rax            # save result
        call    syscall_munmap_chk       # munmap(ptr, len)
        mov     rax, [rsp+48]            # restore result
        add     rsp, 160                 # restore stack
        ret                              # return
```

20.3.3 seed

Given a pointer to a seed buffer, load the encoded value and return it via rax.

This just loads the header, loads all the sizes, and then calls into `frags`.

TODO: Document this.

```
seed:
        mov     r12, rdi
        push    r12
        call    seed.inner
        pop     r12
        ret
seed.inner:
        mov     rcx, [r12]
        lea     rsi, [r12 + 40]
seed.holeloop:
        test    rcx, rcx
        jz      seed.sizes
        xor     edi, edi
        push    rsi
        push    rcx
        push    r12
        call    mkpin
        pop     r12
        pop     rcx
        pop     rsi
        ppush   rax
        dec     rcx
        jmp     seed.holeloop
seed.sizes:
        mov     rcx, [r12 + 8]
seed.sizeloop:
        test    rcx, rcx
        jz      seed.bigs
        mov     r8, [rsi]
        sub     r15, 8
        mov     [r15], r8
        add     rsi, 8
        dec     rcx
        jmp     seed.sizeloop
seed.bigs:
        mov     rcx, [r12 + 8]
seed.bigloop:
        test    rcx, rcx
        jz      seed.words
        dec     rcx
        mov     r8, [r15 + rcx*8]
        mov     rdi, r8
        push    rcx
        push    r8
        push    r12
        call    reserve
        pop     r12
        pop     r8
        mov     rdi, rax
        mov     rcx, r8
        rep     movsq
        push    rsi
        mov     rdi, r8
        call    claim
        pop     rsi
        pop     rcx
        mov     [r15 + rcx*8], rax
        jmp     seed.bigloop
seed.words:
        mov     rcx, [r12 + 16]
seed.wordloop:
        test    rcx, rcx
        jz      seed.bytes
        mov     rax, [rsi]
        push    r12
        call    mkword
        pop     r12
        sub     r15, 8
        mov     [r15], rax
        add     rsi, 8
        dec     rcx
        jmp     seed.wordloop
seed.bytes:
        mov     rcx, [r12 + 24]
seed.byteloop:
        test    rcx, rcx
        jz      seed.frags
        movzx   r8, byte ptr [rsi]
        sub     r15, 8
        mov     [r15], r8
        inc     rsi
        dec     rcx
        jmp     seed.byteloop
seed.frags:
        mov     r8, rsi
        shr     r8, 3
        shl     r8, 3
        mov     rdx, [r12 + 24]
        mov     rdi, rdx
        add     rdi, [r12 + 0]
        add     rdi, [r12 + 8]
        add     rdi, [r12 + 16]
        shl     rdx, 61
        shr     rdx, 58
        mov     rcx, [r12 + 32]
        call    frags
        mov     rax, [r15]
        mov     rcx, [r12]
        add     rcx, [r12 + 8]
        add     rcx, [r12 + 16]
        add     rcx, [r12 + 24]
        add     rcx, [r12 + 32]
        lea     r15, [r15 + rcx*8]
        ret
```

20.3.4 frags

Basically, the just loads a certain number of tree fragments, and appends each one to the environment.

```
rdi - ARG - The total size of the environment table.
rsi - ARG - The environment (an array of plan values).
rdx - ARG - The number of used bits in the current input word.
rcx - ARG - The number of tree fragments to load.
r8  - ARG - A pointer to the current input word.
r10 - TMP - The number of fragments which have been read
r9  - TMP - Misc
```

This basically just calls `frag()` in a for loop, but the reference width may grow each time, so we need to recalculate that for each fragment.

```
frags:                                      # rdi=en rsi=e rdx=boff r8=wp rcx=n
        push    rbx
        push    rbp
        push    r12
        xor     rbx, rbx                    #   i=0 in rbx
        mov     rbp, rdi                    #   en in rdi
        mov     r12, rcx                    #   n in r12
frags.loop:                                 # continue:
        cmp     rbx, r12                    #   if (i >= n)
        jae     frags.ret                   #     break
        lea     rdi, [rbp-1]
        lzcnt   r9, rdi
        mov     rdi, 64
        sub     rdi, r9                     #   bitsz = 64 - lzcnt(sz-1)
        mov     rsi, r15                    #   rsi = sp
        call    frag                        #   frag(bitsz, rsi)
        inc     rbp                         #   i++
        inc     rbx                         #   en++
        jmp     frags.loop                  #   contiue
frags.ret:                                  # break:
        pop     r12
        pop     rbp
        pop     rbx
        ret                                 #   return
```

20.3.5 frag

This is a recursive function that just loads a single bit-packed tree from the input. It will consume a certain number of bits, and the result will be pushed on the PLAN stack.

All this function does is decode these inputs and uses them to construct a lazy function call:

```
(((head arg1) arg2) arg3)
```

The logic here is pretty straightforward, and it's not that much code, but it is quite a lot to understand since there is a lot of state.

```
ARG rdi - refSz (bitSz of a reference into the environment)
ARG rsi - table (environment)
ARG rdx - ub (the number of bits that have been used in the current word).
ARG r8  - fp (pointer into the current word of the input buffer).
TMP rcx - temporary, or width-of-width
TMP r9  - temporary, or number of arguments.
TMP r10 - the next word from the input.
```

The arguments are essentially state that is threaded through the entire process. We don't write these to the stack except to avoid clobbing when calling mkapp().

The tricky bits here are just the decoding of the bit-packed input. The best way to understand this code is to first understand the format, and then work through this logic on paper with some small examples.

A couple of finer points to keep in mind:

- Closures sizes up to 2**32 (or something like that) fit in a single word. By not supporting that, the initial data for each node always fits in 64 bits, which means that all of our bit-decoding logic can work with registers.
- However, we need to read 64 bits of data at a bit-offset. In order to keep things simple and have less state, we just do this every time:

  ```
  word = (p[0]<<o) | (p[1]<<o)
  ```

  This means that we are re-reading the same words multiple times, but this isn't a major concern because of caching, and it keeps the complexity tractable.
- The handling of the zero-parameters case adds little code, but is a little bit tricky.

```
# state: rdi=refsz rsi=env rdx=used r8=ptr
# local: r10=word rcx=tmp r9=params
frag:                                       # frag:
        push    r12                         #   avoid clobbering r12
        call    frag.recur                  #   enter recursive logic
        pop     r12                         #   restore r12
        ret                                 #   return
frag.recur:                                 # recur:
        sub     rsp, 32                     #   frame=(params, ptr, refsz, env)
        mov     [rsp+16], rdi               #   save refsz
        mov     [rsp+24], rsi               #   save env
        mov     r10, [r8]                   #   word = ptr[0]
        test    rdx, rdx                    #   if (used == 0)
        jz      frag.size                   #   goto size
        mov     rcx, rdx                    #   (shr only words with rcx)
        shr     r10, rcx                    #   word <<= used
        mov     rcx, 64                     #
        sub     rcx, rdx                    #   remain = 64-used
        mov     r9, [r8 + 8]                #   next = ptr[1]
        shl     r9, rcx                     #
        or      r10, r9                     #   word = (word | next<<remain)
frag.size:                                  # size:
        xor     r9, r9                      #   params = 0
        tzcnt   rcx, r10                    #   szsz = ctz(word)
        shr     r10, rcx                    #
        shr     r10, 1                      #   word >>= szsz+1 (advance)
        inc     rdx                         #   used++ (for 0 case)
        test    rcx, rcx                    #   if (!szsz)
        jz      frag.head                   #     goto head (0 case has no size)
frag.args:                                  # args:
        dec     rdx                         #   used-- (undo inc)
        add     rdx, rcx                    #
        add     rdx, rcx                    #   used += szsz*2
        dec     rcx                         #   swid = (szsz-1)
        xor     r9, r9                      #
        bts     r9, rcx                     #
        dec     r9                          #
        and     r9, r10                     #   params = word&((2**swid)-1)
        bts     r9, rcx                     #   set high bit
        shr     r10, rcx                    #   word>>swid
frag.head:                                  # head: (r10=word r9=params)
        xor     rcx, rcx                    #
        bts     rcx, rdi                    #
        dec     rcx                         #
        and     rcx, r10                    #   rix = word & ((2**refsz)-1)
        mov     rcx, [rsi + rcx*8]          #   ref = env[rix]
        ppush   rcx                         #   push rcx
        add     rdx, rdi                    #   used += refsz
        cmp     rdx, 64                     #   if (used < 64)
        jb      frag.loop                   #     goto loop
        sub     rdx, 64                     #   used -= 64
        add     r8, 8                       #   ptr++
frag.loop:                                  # loop:
        test    r9, r9                      #   if (params == 0)
        jz      frag.break                  #     break
        mov     [rsp], r9                   #   save params
        call    frag.recur                  #   recur()
        mov     [rsp+8], r8                 #   save ptr
        call    mkapp                       #   mkapp()
        mov     r9, [rsp]                   #   restore params
        mov     r8, [rsp+8]                 #   restore ptr
        mov     rdi, [rsp+16]               #   restore refsz
        mov     rsi, [rsp+24]               #   restore env
        dec     r9                          #   params--
        jmp     frag.loop                   #   continue
frag.break:                                 # break:
        add     rsp, 32                     #   pop frame
        ret                                 #   return
```

21 Parallelism

PLAN has no dependency on a C library. That means we need to implement all the basic threading and parallelism primitives ourselves.

21.1 Mutex

Mutexes are a single 32 bit integer in memory and mutex handling functions take a pointer to that memory address. That piece of memory can be in one of three states:

```
.equ    MUTEX_UNLOCKED, 0                   # Mutex is available
.equ    MUTEX_LOCKED, 1                     # Mutex is locked, no waiters
.equ    MUTEX_LOCKED_WAITERS, 2             # Mutex is locked with waiters
```

21.1.1 mutex_init

A mutex is initialized by zeroing out its int32 value.

```
.global mutex_init
mutex_init:                                 # mutex_init(rdi=int*):
    mov DWORD PTR [rdi], MUTEX_UNLOCKED     #   *rdi = MUTEX_UNLOCKED
    ret                                     #   return
```

21.1.2 mutex_lock

Locking a mutex has a fast path, where you can locally just atomically change a mutex's value from unlocked (0) to locked (1). When the mutex is uncontested, taking the mutex is very, very fast.

When you try to take a mutex that's already locked, you have to worry about whether if you're the first contestor. The implementation falls through to `wait_loop` which checks the `cmpxchg`ed value of the mutex and makes sure it's `MUTEX_LOCKED_WAITERS` before calling the kernel futex implementation.

In `check_unlocked`, after futex wakeup or failed state change from `wait_loop`, you try to acquire the lock again. You always try to set to `MUTEX_LOCKED_WAITERS` because we know other threads might be involved and we'll have to signal to them.

```
.global mutex_lock
mutex_lock:                                 # mutex_lock(rdi=int*):
        mov     eax, MUTEX_UNLOCKED         #   eax = expected value (0)
        mov     ecx, MUTEX_LOCKED           #   ecx = new value (1)
        lock cmpxchg DWORD PTR [rdi], ecx   #   try atomically change 0 to 1
        je      mutex_lock.success          #   if success, we're done
        push    r12                         #   slow path when locked
        mov     r12, rdi                    #   save the mutex pointer
mutex_lock.wait_loop:                       # wait_loop:
        cmp     eax, MUTEX_LOCKED_WAITERS   #   if already marked with waiters
        je      mutex_lock.futex_wait       #   then go wait for kernel futex
        mov     eax, MUTEX_LOCKED           #   eax = expected value (1)
        mov     ecx, MUTEX_LOCKED_WAITERS   #   ecx = new value (2)
        lock cmpxchg DWORD PTR [r12], ecx   #   try atomically change 1 to 2
        jne     mutex_lock.check_unlocked   #   if failed, check if unlocked
mutex_lock.futex_wait:                      # futex_wait:
        mov     rdi, r12                    #   restore rdi
        mov     edx, MUTEX_LOCKED_WAITERS   #   edx = 2
        call    futex_wait_private          #   futex_wait(mutex*, WAITERS)
mutex_lock.check_unlocked:                  # check unlocked:
        mov     eax, MUTEX_UNLOCKED         #   eax = expected value (0)
        mov     ecx, MUTEX_LOCKED_WAITERS   #   ecx = new value (2)
        lock cmpxchg DWORD PTR [r12], ecx   #   try atomically change 0 to 2
        jne     mutex_lock.wait_loop        #   if failed, go back to wait loop
        pop     r12                         #   lock acquired, restore
mutex_lock.success:                         # success:
        ret                                 #   return
```

21.1.3 mutex_unlock

When we unlock, we check whether the old value was `MUTEX_LOCKED` so we can tell the kernel to wake a different waiter if necessary. Note that we unlock the mutex completely before calling into futex; that's expected for this algorithm. Compare with how `mutex_lock.check_unlocked` expects a 0 value when woken up.

```
.global mutex_unlock
mutex_unlock:                               # mutex_unlock(rdi=int*):
        mov     eax, -1                     #   -1 to subtract 1
        lock    xadd DWORD PTR [rdi], eax   #   xadd returns old value in eax
        cmp     eax, MUTEX_LOCKED           #   if old value was 1 (now 0)
        je      mutex_unlock.done           #   then no waiters to wake
        mov DWORD PTR [rdi], MUTEX_UNLOCKED #   unlock completely
        mov     edx, FUTEX_WAKE_ONE         #   call futex to wake one waiter
        call    futex_wake_private          #   rdi already contains mutex ptr
mutex_unlock.done:
        ret
```

21.2 Condition Variables

A condition variable is a "struct" of a u64 pointer to a mutex and a sequence variable.

Offset	Description
+0	mutex pointer
+8	sequence var

The sequence variable is a 32-bit value. Technically, this could wrap around but requires `2^32` signals in the tiny window between seq load and the `futex_wait` kernel call, which is physically impossible in our environment.

21.2.1 condition_init

Our condition variables don't follow the `pthread_cv` interface: a paired mutex is selected at condition variable initialization time. This actually simplifies the implementation.

```
.global condition_init
condition_init:                             # condition_init(rdi=cv*, rsi=mx*):
        mov     qword ptr [rdi], rsi        #   [rdi] = mutex pointer
        mov     dword ptr [rdi + 8], 0      #   [rdi+8] = sequence var, 0 or 1
        ret
```

21.2.2 condition_signal

```
.global condition_signal
condition_signal:                           # condition_signal(rdi=cvar*):
        lock    inc dword ptr [rdi + 8]     #   increment number of signals
        lea     rdi, [rdi + 8]              #   wake the sequence var location
        mov     edx, FUTEX_WAKE_ONE         #   wake only one waiter
        call    futex_wake_private          #   call futex wake
        ret                                 #   done
```

21.2.3 condition_broadcast

```
.global condition_broadcast
condition_broadcast:                        # condition_broadcast(rdi=cvar*):
        mov     r9d, 1                      #   increment by one
        lock xadd dword ptr [rdi + 8], r9d  #   prev value returned in eax
        inc     r9d                         #   r9d is now the new value.
        mov     r8, [rdi]                   #   requeue to the mutex
        lea     rdi, [rdi + 8]              #   wake on the sequence var
        jmp     futex_cmp_requeue_private   #   tail call futex requeue
```

21.2.4 condition_wait

```
.global condition_wait
condition_wait:                              # condition_wait(rdi=cvar*):
        push    r12                          #   save register
        mov     r12d, [rdi + 8]              #   save current sequence var
        push    rdi                          #   save input cvar*
        mov     rdi, [rdi]                   #   set up mutex param
        call    mutex_unlock                 #   mutex_unlock(cvar->mutex)
        pop     rdi                          #   restore cvar*
        push    rdi                          #   save cvar*
        add     rdi, 8                       #   rdi = cvar->seq
        mov     edx, r12d                    #   edx = sequence we captured
        call    futex_wait_private           #   wait for next
        pop     rdi                          #   restore cvar*
        mov     rdi, [rdi]                   #   rdi = mutex for rest of fxn
condition_wait.wakeup_loop:                  # wakeup_loop:
        mov     eax, MUTEX_LOCKED_WAITERS    #   prepare locked+waiters state
        lock xchg DWORD PTR [rdi], eax       #   lock mutex, mark contended
        test    eax, eax                     #   if mutex was unlocked before
        jz      condition_wait.done          #   then we are done
        push    rdi                          #   save mutex pointer
        mov     esi, MUTEX_LOCKED_WAITERS    #   expect mutex to be locked
        call    futex_wait_private           #   call futex wait
        pop     rdi                          #   restore mutex pointer
        jmp     condition_wait.wakeup_loop   #   continue
condition_wait.done:                         # done:
        pop     r12                          #   restore r12
        ret                                  #   return
```

21.3 Read/Write Locks

Our reader writer locks are use 56-byte structure with the reader and writer structs inlined into the structure.

Offset	Size	Description
+0	unsigned int, 4 bytes	mutex
+4	4 bytes	padding (for 8-byte alignment)
+8	pointer, 8 bytes	readers_cv.mutex pointer
+16	unsigned int, 4 bytes	readers_cv.sequence var
+20	4 bytes	padding (for 8-byte alignment)
+24	pointer, 8 bytes	writers_cv.mutex pointer
+32	unsigned int, 4 bytes	writers_cv.sequence var
+36	4 bytes	padding
+40	unsigned int, 4 bytes	active readers
+44	unsigned int, 4 bytes	active_writer
+48	unsigned int, 4 bytes	waiting_writers
+52	4 bytes	padding (for 8-byte alignment)

Our reader/writer lock currently has a strong reader preference; this could lead to writer starvation. This is fine for now, because the only usage of rwlocks is communication between the buddy allocator and the concurrent garbage collector where the allocator allocating should starve forward progress in collection compared to starving allocation completion on work threads.

21.3.1 rwlock_init

```
.global rwlock_init
rwlock_init:                                 # rwlock_init(rdi=rwlock*):
        push    r12                          #   save r12
        mov     r12, rdi                     #   save rwlock pointer
        call    mutex_init                   #   init mutex
        lea     rdi, [r12 + 8]               #   rdi = &rwlock->readers_cv
        lea     rsi, [r12]                   #   rsi = &rwlock->mutex
        call    condition_init               #   initialize condition var
        lea     rdi, [r12 + 24]              #   rdi = &rwlock->writers_cv
        lea     rsi, [r12]                   #   rsi = &rwlock->mutex
        call    condition_init               #   initialize condition var
        mov     DWORD PTR [r12+40], 0        #   active_readers = 0
        mov     DWORD PTR [r12+44], 0        #   active_writer = 0
        mov     DWORD PTR [r12+48], 0        #   waiting_writers = 0
        pop     r12                          #   restore r12
        ret
```

21.3.2 rwlock_read_lock

```
.global rwlock_read_lock
rwlock_read_lock:                            # rwlock_read_lock(rdi=rwlock*):
        push    r12                          #   save r12
        mov     r12, rdi                     #   save rwlock pointer
        call    mutex_lock                   #   lock the mutex
rwlock_read_lock.wait_loop:                  # wait_loop:
        cmp     DWORD PTR [r12 + 44], 0      #   if active_writer == 0
        je      rwlock_read_lock.no_writer   #   then advance to next step
        lea     rdi, [r12 + 8]               #   rdi = &rwlock->readers_cv
        call    condition_wait               #   wait on readers_cv
        mov     rdi, r12                     #   restore rdi to rwlock
        jmp     rwlock_read_lock.wait_loop   #   try again
rwlock_read_lock.no_writer:                  # done:
        inc     DWORD PTR [r12 + 40]         #   increment readers atomically
        mov     rdi, r12                     #   rdi = rwlock*
        call    mutex_unlock                 #   mutex_unlock(rdi)
        pop     r12                          #   restore r12
        ret
```

21.3.3 rwlock_read_unlock

```
.global rwlock_read_unlock
rwlock_read_unlock:                          # rwlock_read_unlock(rdi=rwlock*):
        push    r12                          #   save r12
        mov     r12, rdi                     #   save rwlock pointer
        call    mutex_lock                   #   mutex_lock(rwlock)
        dec     DWORD PTR [r12 + 40]         #   decrement active readers
        cmp     DWORD PTR [r12 + 40], 0      #   if active readers != 0
        jne     rwlock_read_unlock.done      #   then done
        lea     rdi, [r12 + 24]              #   rdi = &rwlock->writers_cv
        call    condition_signal             #   signal one writer
rwlock_read_unlock.done:                     # done:
        mov     rdi, r12                     #   rdi = rwlock*
        call    mutex_unlock                 #   unlock the mutex
        pop     r12                          #   restore r12
        ret
```

21.3.4 rwlock_write_lock

```
.global rwlock_write_lock
rwlock_write_lock:                           # rwlock_write_lock(rdi=rwlock*):
        push    r12                          #   save r12
        mov     r12, rdi                     #   save rwlock pointer
        call    mutex_lock                   #   lock the mutex
        inc     DWORD PTR [r12 + 48]         #   increment waiting writers
rwlock_write_lock.wait_loop:                 # wait_loop:
        cmp     DWORD PTR [r12 + 40], 0      #   if there are active readers
        jne     rwlock_write_lock.wait       #   then wait
        cmp     DWORD PTR [r12 + 44], 0      #   if there's no writer active
        je      rwlock_write_lock.done       #   then finish up and take lock
rwlock_write_lock.wait:                      # wait:
        lea     rdi, [r12 + 24]              #   rdi = &rwlock->writers_cv
        call    condition_wait               #   wait on writers_cv
        mov     rdi, r12                     #   restore rdi
        jmp     rwlock_write_lock.wait_loop  #   try again
rwlock_write_lock.done:                      # done:
        mov     DWORD PTR [r12 + 44], 1      #   mark writer as active
        dec     DWORD PTR [r12 + 48]         #   decrement waiting writers
        mov     rdi, r12                     #   rdi = rwlock*
        call    mutex_unlock                 #   unlock mutex
        pop     r12                          #   restore r12
        ret
```

21.3.5 rwlock_write_unlock

```
.global rwlock_write_unlock
rwlock_write_unlock:                         # rwlock_write_unlock(rdi=rwlock*):
        push    r12                          #   save r12
        mov     r12, rdi                     #   save rwlock pointer
        call    mutex_lock                   #   lock the mutex
        mov     DWORD PTR [r12 + 44], 0      #   mark writer as inactive
        cmp     DWORD PTR [r12 + 48], 0      #   if waiting_writers == 0
        je      rwlock_write_unlock.signal   #   then signal readers
        lea     rdi, [r12 + 24]              #   rdi = &rwlock->writers_cv
        call    condition_signal             #   signal one writer
        jmp     rwlock_write_unlock.done     #   skip to the end
rwlock_write_unlock.signal:                  # signal:
        lea     rdi, [r12 + 8]               #   rdi = &rwlock->readers_cv
        call    condition_broadcast          #   broadcast to all readers
rwlock_write_unlock.done:                    # done:
        mov     rdi, r12                     #   rdi = rwlock*
        call    mutex_unlock                 #   unlock the mutex
        pop     r12                          #   restore r12
        ret
```

21.4 Barriers

A barrier waits on a value to become 0. Used for count down tasks, usually where there's one waiter who farms out tasks to threads.

21.4.1 barrier_set

```
.global barrier_set
barrier_set:                                 # barrier_set(rdi=barrier*,rsi=cnt):
    lock xchg [rdi], rsi                     #   *barrier = count
    ret                                      #   return
```

21.4.2 barrier_done

```
.global barrier_done
barrier_done:                                # barrier_done(rdi=barrier*):
        mov     eax, -1                      #   decrement by 1
        lock xadd dword ptr [rdi], eax       #   locked perform subtract
        cmp     eax, 1                       #   if we aren't last thread
        jne     barrier_done.done            #   then don't wake anyone
        mov     edx, FUTEX_WAKE_ALL          #   otherwise wake everyone
        call    futex_wake_private           #   call futex
barrier_done.done:
        ret
```

21.4.3 barrier_wait

```
.global barrier_wait
barrier_wait:                                # barrier_wait(rdi=barrier*):
        mov     edx, [rdi]                   #   current number of waiters
        test    edx, edx                     #   if this is zero
        jz      barrier_wait.done            #   then we have don't have to wait
        push    rdi                          #   save barrier*
        call    futex_wait_private           #   call futex_wait
        pop     rdi                          #   restore barrier*
        jmp     barrier_wait                 #   try again
barrier_wait.done:
        ret
```

21.5 Threads

We use the following structure for a thread:

Offset	Size	Description
+0	unsigned int, 4 bytes	tid
+4	4 bytes	padding
+8	unsigned int, 4 bytes	futex exit_flag
+12	4 bytes	padding
+16	pointer, 8 bytes	function pointer to run
+24	void*, 8 bytes	argument to pass to function
+32	void*, 8 bytes	return value storage
+40	void*, 8 bytes	stack pointer
+48	size_t, 8 bytes	stack size

21.5.1 thread_create

Thread create takes three arguments: A pointer to empty memory for the thread struct in rdi, a function pointer to run on the thread in rsi, and an argument to pass to that function pointer in rdx.

The one thing that's subtle here is the handling of the start path. So `clone3` will return 0 to the child thread, and the pid to the parent thread. When we jump into the child thread, we have to immediately set up the stack by creating a base stack frame and make sure we're stack aligned.

```
.equ    CLONE_FLGS, CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD
.equ    EIGHTMB,    8 * 1024 * 1024

.global thread_create
thread_create:                               # thread_create(rdi, rsi, rdx):
        push    rbp                          #   frame header
        mov     rbp, rsp                     #   establish frame
        push    r12                          #   save callee-saved
        mov     r12, rdi                     #   r12 = thread*
        mov     qword ptr [r12], 0           #   tid = 0 (init structure)
        mov     qword ptr [r12 + 8], 0       #   futex exit_flag = 0
        mov     qword ptr [r12 + 16], rsi    #   function pointer to run
        mov     qword ptr [r12 + 24], rdx    #   argument to function pointer
        mov     qword ptr [r12 + 32], 0      #   return value storage
        mov     qword ptr [r12+48], EIGHTMB  #   stack size
        mov     rax, SYS_MMAP                #    set up MMAP call for stack
        mov     rdi, 0                       #   addr = NULL
        mov     rsi, [r12 + 48]              #   size = stack_size
        mov     rdx, PROT_READ|PROT_WRITE    #   prot = read/write
        mov     r10, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK
        mov     r8, -1                       #   no file descriptor backing
        mov     r9, 0                        #   offset = 0
        syscall                              #   mmap(...)
        cmp     rax, 0                       #   if return mmap < 0
        jl      thread_create.error          #   then handle error
        mov     [r12 + 40], rax              #   stack_base = mmap addr
thread_create.do:                            # do:
        mov     rsi, [r12 + 40]              #   base child stack pointer
        add     rsi, [r12 + 48]              #   base + size = top of stack
        mov     rax, SYS_CLONE               #   SYS_CLONE
        mov     rdi, CLONE_FLGS              #   set clone flags
        xor     rdx, rdx                     #   ptid = NULL
        xor     r10, r10                     #   ctid = NULL
        xor     r8, r8                       #   newtls = 0
        syscall                              #   clone(flags, child_stack, ...)
        cmp     rax, 0                       #   test clone return value
        jl      thread_create.error          #   rax < 0: error
        je      thread_create.start          #   child path (clone returns 0)
        mov     [r12], rax                   #   parent path:store rax=chld tid
        pop     r12                          #   restore r12
        mov     rax, 0                       #   return success
        leave                                #   restore stack
        ret                                  #   return
thread_create.error:                         # error:
        leave                                #   restore stack
        ud2                                  #   kill process
thread_create.start:                         # start:
        xor     rbp, rbp                     #   clear frame pointer for new thrd
        and     rsp, -16                     #   ensure 16-byte stack alignemnt
        mov     rdi, [r12 + 24]              #   rdi = arg to function pointer
        push    r12                          #   save r12
        call    [r12 + 16]                   #   call function pointer
        pop     r12                          #   restore r12
        mov     [r12 + 32], rax              #   save function result
        mov     eax, 1                       #   set exit mark
        lock xchg dword ptr [r12 + 8], eax   #   mark thread as exited
        lea     rdi, [r12 + 8]               #   rdi = atomic exit_flag
        mov     edx, FUTEX_WAKE_ONE          #   wake a single waiter
        call    futex_wake_private           #   futex_wake()
        mov     rdi, rax                     #   function return val as exit code
        call    syscall_exit                 #   and call exit
```

21.5.2 thread_join

Thread joining takes a pointer to a thread structure and optionally a pointer to memory to return a value from the thread.

```
.global thread_join
thread_join:                                 # thread_join(rdi, rsi):
        cmp     dword ptr [rdi + 8], 0       #   if futex exit_flag is not set
        jz      thread_join.futex_wait       #   then wait for thread
        test    rsi, rsi                     #   if no result pointer
        jz      thread_join.munmap_stack     #   then skip setting result
        mov     rax, [rdi + 32]              #   get result
        mov     [rsi], rax                   #   set result
thread_join.munmap_stack:                    # munmap_stack:
        mov     rdi, [rdi + 40]              #   rdi = stack pointer
        mov     rsi, [rdi + 48]              #   rsi = stack size
        mov     rax, SYS_MUNMAP              #   munmap
        syscall                              #   munmap(rdi, rsi)
        ret                                  #   return
thread_join.futex_wait:                      # futex_wait:
        push    rdi                          #   save rdi
        push    rsi                          #   save rsi
        lea     rdi, [rdi + 8]               #   wait on futex exit_flag
        mov     edx, 0                       #   expect 0
        call    futex_wait_private           #   futex_wait()
        pop     rsi                          #   restore rsi
        pop     rdi                          #   restore rdi
        jmp     thread_join                  #   try again after futex wake
```

21.6 Futex Helpers

Calling the futex syscall is repetitive across our parallelism code. We use helpers that otherwise work with kernel syscall calling conventions.

21.6.1 futex_wait_private

`rdi` is a u32 memory location to wait on, while `edx` is the expected value of that memory location for early futex return.

```
futex_wait_private:                          # futex_wait_private(rdi, edx):
        mov     rax, SYS_FUTEX               #   rax = futex syscall
        mov     esi, FUTEX_WAIT|FUTEX_PRIVATE_FLAG
        xor     r10, r10                     #   timeout = NULL (wait forever)
        xor     r8, r8                       #   r8 unused
        xor     r9, r9                       #   r9 unused
        syscall                              #   rdi and edx already set
        ret                                  #   return
```

21.6.2 futex_wake_private

`rdi` is a u32 memory location to wake, while `edx` is the number of waiters to wake up.

```
futex_wake_private:                          # futex_wake_private(rdi, edx):
        mov     rax, SYS_FUTEX               #   rax = futex syscall
        mov     esi, FUTEX_WAKE|FUTEX_PRIVATE_FLAG
        xor     r10, r10                     #   r10 ununused
        xor     r8, r8                       #   r8 unused
        xor     r9, r9                       #   r9 unused
        syscall                              #   rdi and edx already set
        ret                                  #   return
```

21.6.3 futex_cmp_requeue_private

Requeues waiters from one memory location to another.

`rdi` is a u32 memory location to wake. `r8` is the memory location to requeue all waiters other than the one we wake. `r9d` is the check sequence number.

```
futex_cmp_requeue_private:                   # futex_cmp_requeue_private(rdir8r9)
        mov     rax, SYS_FUTEX               #   rax = futex syscall
        mov     esi, FUTEX_CMP_REQUEUE|FUTEX_PRIVATE_FLAG
        mov     edx, 1                       #   wake exactly one waiter
        mov     r10, 0x7fffffff              #   requeue all other waiters
        syscall                              #   r8 and r9d already set
        ret                                  #   return
```

22 Proposals

22.1 Seed Format Changes

22.1.1 Constant Reference Width

Right now, after each fragment is loaded, it is pushed to the stack, which increases the maximum possible reference number, which potentially increases bit-width of each reference.

Instead of doing this, we could just always operate with the maximum reference size required by the file. This would increase the size of the seed file somewhat, but it would simplify decoding, and it might actually improve the size of compressed seeds, since it makes repeated patterns have a more predictable shape.

22.1.2 Encoding Graphs

If we use a constant reference width, that opens up the potential for seed to be used to encode recursive values, like with LETREC.

To implement this, you would just pre-populate each fragment as a blackhole'd thunk, and then you would update the thunk with the actual value as each fragment is read.

The biggest downside here would be needing to validate that all references are backwards, when loading seeds which are meant to be DAGs. This is fairly easy, however, since a validating loader already needs to keep track of the maximum valid reference.

22.1.3 Smaller Words

The current format encodes fixed-width numbers as either Word64 or Word8, but this wastes a lot of space.

We can actually add 32-bit and 16-bit words to this set without changing the size of the header, or breaking alignment.

This would decrease the size of seeds, and add only a tiny bit more complexity to the decoder.

The trick is to use a smaller word to encode the count of the smaller words. There are 2**32 possible 32-bit words, which is one too big to fit in a 32-bit word, but if you exclude 16-bit words from that set, then it always fits. The same logic applies for 16-bit words, and then we use a 16-bit words to represent the number of bytes.

A 32-bit word and two 16-bit words still weights 64 bits, so the rest of the structure is unchanged.

```
struct header {
    uint64_t numArgs;
    uint64_t numBigs;
    uint64_t n64;
    uint32_t n32;
    uint16_t n16;
    uint16_t n8;
    uint64_t numFrags;
}
```

22.2 Pointer Tagging Optimizations

There are a number of potential changes to this scheme which are worth exploring.

1. The second bit being set for all non-nat values is no longer necessary, since we have a sufficiently efficient way to ask if a number is a natural using rorx and cmp.
2. The tag bits can be shifted all the way to the left, to provide a bit more extra data.
3. The tag on natural numbers should maybe include a size. This would remove the need for the size-metadata on the heap object for small numbers, it would make fetching the size cheaper, and would make comparisons of small bignats faster. OTOH, it would make constructing nats a little bit more expensive and complicated. This requires careful evaluation before proceeding.
4. Natural numbers should include a tag bit which indicates whether or not they refer to a nat that lives within a frozen pin. This will give us the ability to keep unfrozen nats off of the moving heap, which avoids needing to copy them all the time, and also makes their locations stable.
5. The size of the tag and size information on closures could be expanded to 5 bits (max direct tag/size would be 30, this would hurt legibility in a debugging but would make switching on bigger ADTs possible without an indirection and would remove the need to store size metadata on bigger closures.

22.2.1 First Alternative

The result of these changes may end up looking something like this, but the specifics are TBD:

```
u63 0$$$$$$$$$$$$$$$
NAT 100001fsssssssss (f=frozen, s=size)
PIN 10001000000000cm (c=hascrc32, m=ismegapin)
LAW 1001000000000000
CLZ 101000tttttzzzzz (t=tag, z=size)
THK 1100000000000000
```

The idea with size of the nat tag is to include 9 bits of size information, which is enough to have known sizes for bignats up to 64 bytes.

- Since bit-sizes 0-63 are always direct, this allows us to have directly known sizes for sizes up to 574 bits.

- However, we need a special value to indicate "too big". We will use all ones for that case.

- Therefore, the procedure for extracting the size information would be to extract the size bits (via a mask), and then add 64.

- Otherwise, you can find the bitsize by extracting the s bits, and then subtracting 63

22.2.2 Second Alternative

Another alternative, which continue to allow the high-byte to uniquely define heap reference types, could be:

```
u63 0XXXXXXXXXXXXXXX
NAT 1000000(xxxxxxxx small indirect nat, (x+8=size)
NAT 1000001(xxxxxxxx big indirect nat, (x+8=size)
FAT 1000010(xxxxxxxx pinned nat, (x+8=size)
PIN 10001000000000cnn (c=bacrc32, n=somepin)
LAW 1001000000000000
CLZ 101000000ttttxxxx (t=tag, x=size)
THK 1100000000000000
```

22.3 Buffers / FATs

Right now, large, unpinned natural numbers live directly on the moving heap, but this has a number of downsides:

- This requires that this data be copied on every GC which is expensive for large buffers.
- We want to be able to store generated code for laws in natural numbers, and this code will need to not be moved by garbage collection.
- We want to be able to allocate temporary buffers in order to work with syscalls, and we need to be sure that those are not moved.

Here is a proposed sequence of proposals for introducing this concept and then integrating it deeper and deeper into the system.

22.3.1 xbuf

GC1 already supports top-level nat allocations, and the bucket-marking system used for all GC2 references should already work for marking such nats, so a basic version of this should be possible with very little change.

The heap layout and pointer tagging scheme would be totally conventional, just a normal NAT allocation, just allocated in GC2. Just introduce a few XPLAN primitives:

- `xbuf : Nat -> Nat` constructs a buffer of byte-size(n) as a Nat which is allocated as a top-level allocation at a fixed location.

  This should crash if asked to allocate a size smaller than 8, the actual allocation should be one byte bigger than requested and the high byte should be set to 1 (making this a Bar), so that normal in-place NAT operations can be used to manipulate this.

- `xptr : Nat -> Nat` converts from a number in fixed memory (either a fixed nat, or a nat that lives inside of a pin), and converts it into a pointer.

  This can be used with system calls to read data into a nat (even at an offset), or to write data from a nat (for example, to serialize data directly from pinned nat without needing to make a copy.

  This can also be used if we dynamically generate code, to get a code pointer that can be attached as the implement for a pin or a law.

- `xtouch: a -> b -> b` is used to guarantee that a temporary buffer is not freed until we are doing using a raw pointer into it. This is equivalent to Seq, but with the guarantee that it will not be optimized away, even if an optimizer can prove that the value has already been evaluated. (right now, there is not difference since we do not optimize Seq, but we will eventually).

Significant RPN tests should be added which demonstrate that this all works correctly and is correctly collected.

This should be sufficient for basic usage, but I have a number of operational problems which we become a problem under heavier production use.

22.3.2 GC1 Collection

Heavy use of simple fixed buffers will result in a much higher allocation rate in GC2, and keeping this allocation rate low (through pin compaction) is a central pillar of our design.

However, fixed nats are not pins and cannot be referenced from pins or shared between actors, so they should in theory be collectable directly by GC1.

Here's an outline of how such a design could work:

- First, give fixed nats (FATs) their own pointer tag. There is enough bit space for this in the high byte, and the (nat/pnat) checks are already range-checks, so they should be able to support multiple types at once without any change.
- Second, change the layout of FATs so that they have a mark bit and an intrusive linked list. GC2 still requires a GC header on the outside, so the layout should be something like:

  `| GCHDR | mark | next | GCHDR | word | word | word`

- Reserve the first stack slot of the shadow (r15) stack to be a pointer to the FAT list. Every FAT allocation prepends itself to this list.
- During GC1 collection, when we encounter a reference to a FAT, set the mark.
- After GC1 collection, traverse the linked list of FATs and free/delete any node which is not marked.

This scheme should add very little overhead, and the result is that FATs are freed promptly instead of clobbing up the heap between GC2 collections.

It's important the think through the synchronization issues between GC1 and GC2 collections very carefully, but intuitively this scheme should be fine:

- When GC2 is not running there is no race issue.
- During initial mark collection, this scheme changes nothing.
- After initial mark collection, all old/reachable things are marked.
- During sweep, all allocations are allocated marked.
- The sweep traversal uses a lock that is shared with free(). So, if GC1 frees a FAT while GC2 is sweeping, that should not be a problem.

22.3.3 BigNats always FAT

In addition to the explicit xbuf operation, all large numbers should be allocated as FATs. This keeps the GC1 heaps small, which improves GC frequency, and avoids copying these large binary objects on every collection.

The conventional cut-off for this is 64 bytes.

This is conceptually trivial, but will be somewhat annoying in practice because it breaks the `reserve/claim` protocol. However, if we wait until we add support for nats that can be truncated in place, this problem should go away, since that change will avoid the need for this special dance altogether.

22.4 Actors

Our GC architecture is designed to be thread-safe, but we currently only use it with one thread at a time. Getting things to *actually* work with multiple threads at once will likely require quite a lot of effort.

22.5 Specialized Executioners

In addition to the unknown executioners or thunks, we should create specialized executioners for saturated (and oversaturated) calls to known operations.

Something like this, for example:

```
{THUNK(5), xknown2, &addop, Add, x, y}

{THUNK(7), xknown2over, &addop, Add, 0, 1, 2, 3}
```

This would eliminate a lot of overhead, and wouldn't add all that much complexity. However, we will need a compiler before we can make use of this.

22.6 Freelists in Persistence

Right now, our persistence file format is append only, with garbage collection of regions only punching holes in the file, zeroing them and freeing the space. But this doesn't allow reuse of this address space, and very long running persistence files will run out.

After a garbage collection pass, we have a map of what regions of the file are free. We should write this map to the file and allocate from this freelist.

22.7 Register Reform

First, treating the C stack as GC roots will allow us to get rid of the r15 stack, freeing up another register. It might be worth using rbp as the heap pointer and rbx as the heap end, which would free up all of the numbered registers.

Second, the convention of using r12 as a scratch registers conflicts very badly with it's usual role as a callee-saved register, and another register should replace it for that purpose: maybe r11?

Third, we gain a lot of value from using custom calling conventions in a lot of places, but having reliable callee-saved registers would make a lot of things easier. These rules must be respected everywhere, or else you can't rely on them anywhere.

Together, this changes would make r12, r13, r14, and r15 available as callee-saved registers.

22.8 Killing the Shaddow Stack

Killing the shadow stack would free up a register, and it would also mean that each actor has one less region to worry about.

In particular, the Erlang process model has a single allocation area per-actor which contains the stack and the heap, and they grow in opposite directions. This makes allocating a new actor super cheap, and makes he per-actor memory footprint very small for small/helper actors.

However, right now we use the RPN/shadow stack quite heavily. Most uses are just for register spilling, but we also use it as an actual stack in quite a few places. All of these systems would need to be reworked somehow.

- In the actual RPN debugger, used for the whole test suite. For the basic tests, we can probably find some reasonable way to write these tests directly in C/C++.

  For more complex tests, we can probably just use seeds / planlisp / Sire.

- The seed loader uses the stack to represent environments, and also as a logical stack for mkapp. The former could just use a single explicit stack-frame, and the latter can be rewritten to use normal register conventions.
- The graph reduction engine uses the shadow stack to handle oversaturation. I'm not sure what the alternative would be, but it should be possible.
- In Judge, the environment and all nested sub-expressions are explicitly laid out on the stack. This will all go away with the new template expansion approach.
- There are likely a few more uses that have been overlooked by the above list.

One other major complication is that scanning the C stack is not at all safe. And this is going to make the interface between C/asm even trickier. We somehow need to avoid scanning C frames. We will either need an explicit roots system or to avoid GC1 allocation from C code at all.

22.9 Faster Evaluation Preludes

At present, almost all primops begin with a prelude which evaluates all of the strict arguments. Something like this:

```
# opcopy: rdi=cnt rsi=sof rdx=dof rcx=src r8=dst
opcopy:                                    # opcopy:
        NAT     rdi, rsi, rdx, rcx, r8     #   eval+cast cnt
        ENAT    rsi, rdi, rdx, rcx, r8     #   eval+cast sof
        ENAT    rdx, rsi, rdi, rcx, r8     #   eval+cast dof
        EVAL    rcx, rsi, rdx, rdi, r8     #   eval src
        EVAL    r8,  rsi, rdx, rcx, rdi    #   eval dst
fastcopy:
        ...
```

These evaluation macros `EVAL` and `ENAT` are optimized to do as little work as possible in the cases where the routine is not given a thunk. And they accomplish this by only flushing to the stack when *actually* given a thunk.

This works by examining each argument one by one. If it is a thunk everything is flushed to the stack, the relevant values are evaluated, and everything is restored from the stack. So, for the above example, if we are given 5 thunks, we will do something like 20 memory writes and 20 memory reads.

Unfortunately, in practice, the routines are almost always given thunks. The only time when we would be given something *besides* a thunk is when a law body passes in a hard-coded constant to a function. This happens, but it is very much not the common case.

Once we have an optimizing law compiler, law judgment itself will do a lot of evaluation directly before deferring to the graph reduction engein, and this will greatly increase the set of places where things besides thunks are passed in.

But such an optimizing law compiler should *also* be able to just perform the requisite evaluations itself, and then directly call into the fast path (`fastcopy` in this case), avoiding the entire prelude.

Given this combination of factors, it might be significantly more efficient to just flush everything to the stack, evaluate each item one-by-one, and the restore everything. In this case, the memory options would be reduced to 10 writes and 10 reads, which is significantly less work:

1. Save all five arguments to the stack.
2. Evaluate each argument, replacing the stack slot with the result.
3. Restore all of the registers from the stack.

And then, when using a more sophisticated judgment strategy, we would instead do the evaluation in the caller, and then directly invoke the fastpath, skipping this entire prelude.

22.10 Exit Segfaults

The code seems to segfault on `CTRL-C`, but only on the first run. What causes this?

22.11 GC2 Under Pressure

What's the best way to handle heavy load of GC2? Since our abstraction hides the memory hierarchy, having any sort of hard-limits on the size of data in each region is not great.

How can we avoid needing to make the buddy allocator resizable?

How can we make sure that heavy use of GC2 swaps, instead of killing the whole process?

22.12 GC2/GC3 Mutation

One of the invariants that we rely on in order to have a concurrent garbage collector without any write barriers is the complete lack of mutation in GC2 and GC3.

Because of the design of our system, this invariant is fairly easy to maintain. However, there are a few cases where it would be nice to violate it.

The big question is, can we sneak in these exceptions without losing the properties which make concurrent GC tractile?

Here are the edge-cases that would be nice to introduce:

- Redirections: If a GC2 pin is persisted, we would like to redirect all other references to the on-disk variant.
- Redirections: If we discover via hashing that two pins are identical, in either GC2 or GC3, we would like be able to redirect one to the other in order to avoid the duplication.
- Lazy Hashing: It would be nice to be able to wait to calculate the cryptographic hash of pins in GC2 until they are actually needed.
- Codegen: Since pins in GC2 and GC3 have stable locations, we can avoid needing to have a "constants" table for generated code, and instead generate machine code which directly references values and code, by raw pointer.

  For example, to constants or other code within the same pin. But if we need to generate the code *before* the pin is constructed, then we don't can't know the memory-locations of these values in advance.

  It would make more sense to figure out in advance how big the generate code would be, then allocate the pin with the code zeroed, then actually fill in the code.

- Moving code from GC2 to GC3. Once we attach generated code to pins and laws, the generated code will somehow need to be updated in order to patch-up the pointers to point to new locations.
- Compaction: It would be nice to be able to re-organize the heap on the fly, to reclaim parts of the address space and reduce fragmentation.

  However, this would require some combination of the above things in order to work well.

Some of this seems like it should be possible, but these things all need to be worked through in incredibly detail before we can conclude that this is viable.

For example, If we just constructed a pin in GC2 and no other code has reference to it, then mutating NAT that contains the code should not cause any issues. Since only binary data is being mutated, no new references can be introduced into the GC graph, so the concurrent GC should simply not care at all.

Similarly, If we pre-populate all cryptographic hashes within GC2 with a placeholder, then filling in that value later is also just a binary change. There are synchronization issues to think through, however.

This is *NOT TRUE* for GC3. Since we need to care about synchronization and atomicity invariants as well, even in-place binary changes are a big problem.

When inserting a redirection from (GC2 -> GC2) or (GC2 -> GC3) the thread performing the indirection necessarily has a reference to both things, so no new references can be "discovered" through this process. This also does not introduce any risk of introducing references from GC3 -> GC2, since we would simply ban such redirections.

This two facts *seems* like they could be sufficient to make doing this safe, but we need to tread super, super carefully here.

22.13 Separate Code Section

I've completely punted on this for now, since it introduces a lot of complexity. And that punting should continue for the foreseeable future.

However, ideally, we would have a separate area in GC2 and in GC3 for code. This way we don't have to map all of our data into executable pages. Also, code can be more compact if it can take advantage of the fact that it is collocated in order to use relative jumps/references, etc.

However! This may introduce so much complexity that it is simply not worth it, despite the significant upsides.

22.14 Lazy Pins

We can't do this yet because this will definitely break compatibility with the Haskell runtime, but one major change which should result in a massive performance improvement is lazy pins and laws.

Right now, whenever we construct a Pin, we normalize the pin and immediately copy everything to GC2.

However, when doing something like inserting a page of JSON into a hitch database in the fulltag demo, we are constructing a huge pile of temporary pins which immediately become garbage.

In the new PLAN standard, pins are *not* guaranteed to be normalized. Also, we can separate out pin construction in GC1 from pin freezing (normalization and movement to GC2). The result will be that all of these ephemeral pins never make it to GC2 and never even need to be copied.

At the end of each step, when we are ready to merge our final change into the resulting state, *only then* do we we recursively freeze all of the pins in the result.

This will massively decrease the amount of garbage allocated into GC2.

And it also means that we don't need to copy all of the data within a pin each time we construct a pin. The new version of a pin can share data with the previous version.

For example, if you do a single insert into the hitchhikers of a hitch node, you do not have to copy the whole table, just the insert into the hitchhikers table and a new root.

I'm not sure if this is advisable in this context, but it even becomes possible to use lazy evaluation to avoid work when performing an iterative series of changes to a pin DAG.