Summer 2020

Functional Programming for Systems Software: Implementing Baremetal Programs in Habit

Donovan Ellison
Portland State University

Recommended Citation
Ellison, Donovan, "Functional Programming for Systems Software: Implementing Baremetal Programs in Habit" (2020). *University Honors Theses.* Paper 947.
https://doi.org/10.15760/honors.970

Functional Programming for Systems Software:

Implementing Baremetal Programs in Habit

By

Donovan Ellison

An undergraduate honors thesis submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science

in

University Honors

and

Computer Science

Thesis Advisor

Mark P. Jones, Ph.D

Portland State University

2020

Contents

Abstract

Programming in a baremetal environment, directly on top of hardware with very little to help manage memory or ensure safety, can be dangerous even for experienced programmers. Programming languages can ease the burden on developers and sometimes take care of entire sets of errors. This is not the case for a language like C that will do almost anything you want, for better or worse. To operate in a baremetal environment often requires direct control over memory, but it would be nice to have that capability without sacrificing safety guarantees. Rust is a new language that aims to fit this role and is relatively similar to C in syntax and functionality. However, it may be worthwhile to branch farther away from C. Functional programming could offer a more productive, verifiably correct development cycle, but implementations of functional languages like Haskell often come with a heavy runtime and no built-in memory management. Thus the inspiration for Habit, a functional programming language specially designed for baremetal programming. This thesis explores Habit and its intermediate language LC to see what benefits could come from developing systems software in a functional language, while simultaneously testing its prototype compiler for bugs, looking for gaps in features, and contributing to the continued development of Habit.

1 Introduction

Software development is tricky. There can be a large gap between what we expect a piece of software to do and the actual properties of the code. The 2020 Open Source Security and Risk Analysis (OSSRA) report audited over 1,200 projects and found 75% had at least one vulnerability while 49% contained high-risk vulnerabilities [1]. Baremetal programs – programs that run directly on top of hardware, such as an operating system – can be particularly hard to verify [2]. Baremetal programs are typically written in C and assembly, since those languages can still interact with and manipulate memory even without a stack or heap. Unfortunately, C and assembly are prone to human error and even minor mistakes could create high-risk vulnerabilities or crash the system altogether [3, 4].

seL4 is a microkernel related to the L4 family of microkernels, composed of ~8,700 lines of C code and ~600 lines of assembly [3]. seL4 is notable as the first operating system with end-to-end verification of its formal specification. This end-to-end verification was especially impressive given the choice of C as an implementation language where there are very few guarantees, especially in regards to null pointers, hanging pointers, and largely unrestricted type casting. To mathematically prove the correctness of the C implementation required ~200,000 lines of Isabelle, a general purpose theorem proving language [3]. Thus the formal verification of C is a massive effort, requiring many more lines of proof than for the implementation itself. These proofs also make it harder to refactor the seL4 implementation, as refactoring also requires adding to or changing large portions of the theorem prover code. So it seems preferable to use a language with memory safety and type correctness guarantees, instead of needing to formally verify those properties in a C implementation.

Using a higher-level language for baremetal programming could have additional benefits aside from reducing formal verification efforts. For seL4, the developers first implemented a prototype of the microkernel in Haskell, which allowed them to reason about the program, identify areas of code that required side effects, and smooth out the design process [3]. The team found the prototyping to be beneficial to productivity, as they were able to design and implement the microkernel in less time than similar projects. Altogether, developing the seL4 Haskell prototype and C implementation took only 2.2 person years (py) worth of effort, whereas similar projects without a Haskell prototype took 4 to 6 py [3]. However, Haskell could only be used for a prototype because GHC, the standard Haskell compiler, has a runtime that is larger than the microkernel itself; the code couldn't be optimized to the same extent; and the language doesn't provide enough control over the memory layout. Given the enormous gains from first prototyping in Haskell, it seems worthwhile to work towards a high-level functional language that can meet the needs of baremetal programming.

At PSU there has been a series of research projects on the use of functional programming languages for baremetal programming. In 2005, an entire OS was built using Haskell, nicknamed House [5]. Haskell cannot do low-level operations, thus House required using the Foreign Function Interface to export code in C that Haskell could not do. These unsafe operations were put into the Hardware monad, or H monad, that was then used to build a kernel, systems programs, and applications. Then, building off this experience, Dr. Leslie wrote her dissertation on the H interface, a library for memory safe baremetal programming in Haskell [6]. However, in both projects there were still performance concerns regarding the Haskell runtime and lack of control over memory layout.

The experience from House and the H interface helped guide the design of the functional programming language, Habit [7]. Habit is partially derived from Haskell, sharing some of its syntax and using a similar type system. Unlike Haskell, Habit has built-in functionality for controlling memory layout and accessing memory locations in a type safe manner. Additionally, Habit lets programmers define the layout and size of data types, as well as the ability to assign names for areas within that type. These built-in features should give the programmer the flexibility to interact with the hardware exactly as they want, while still being type correct and memory safe. Finally, the Habit compiler features several aggressive optimization steps. One of the compilation steps uses the Monadic Intermediate Language (MIL), an intermediate language specifically designed for functional programming languages [8]. MIL retains many properties of a functional programming language and can use those properties to make functional language specific optimizations. Afterwards, the MIL code is translated to LLVM and goes through another optimization process. These features and compilation process aim to bridge the gap between a reliable, high-level language like Haskell and the performance of a language like C.

Research Question: *Can Habit aid the development of reliable, predictable, and portable baremetal programs?*

Since its creation, Habit has seen limited use as a baremetal programming language. Mark Jones wrote a handful of baremetal programs in LC, an intermediate language of the Habit compiler with similar features and syntax. Several of the LC baremetal programs were originally labs in the Low-Level Programming (LLP) course [9]. The page table lab, using the IA32 architecture [10], currently does not have an LC implementation. Thus to answer the research question, this thesis involves implementing a page table in LC based on the associated lab, and comparing it to the C implementation. In the future it may be possible to build a fully functional L4 microkernel from these LC programs, and a page table implementation serves as a step in that direction. Furthermore, continued use of LC provides valuable feedback about compilation issues or features to consider adding.

2 Background

2.1 Habit and LC

LC is an intermediate language for Habit, originally an acronym for Lambda-Case due to it mainly supporting lambda expressions and case constructs [11]. The intent of LC was to ease testing for the monadic intermediate language (MIL), but has since been expanded for use in the Habit compiler. At the time of writing it was preferable to work in LC due to Mark Jones' familiarity with the compiler and ability to work through potential bugs. This leaves LC in a peculiar place, with enough features to produce equivalent code to Habit, but with fewer conveniences. For a full account of Habit's features, see the technical report [7]. In LC there are no type classes, the LC compiler sometimes cannot infer types, and there is no string formatting. This means more verbose and redundant code. However, there are many features in LC that make it worthwhile to use, showcasing some of the core features of Habit as well as additional features added over time.

2.1.1 Numeric Literals

Numeric literals have been expanded upon from Haskell. Integer literals in LC come in the standard integer form (e.g., 2, 100, 1234), but also can be represented in binary and hexadecimal. To create an integer literal in binary requires the `0b` prefix, followed by binary. Integer literals can be represented in hexadecimal as well, but with the `0x` prefix instead.

```
0b1   = 1       0xf  = 15
0b101 = 3       0xab = 171
```

Integer literals may alternatively use a suffix to denote a multiplier.

```
K -> kilo -> 2^10   1K = 2^10 = 1024
                    4K = 2^12 = 4096
M -> mega -> 2^20   2M = 2^21
```

In addition to integer literals, LC provides syntax for fixed width bit vector literals. These literals are represented by the `Bit n` type, where `n` is the number of bits for the associated bit vector. Bit vector literals can be represented in binary or hexadecimal, using the prefix `B` or `X` respectively.

```
B1   -> 1       Xf  -> 1111
B101 -> 101     Xab -> 10101011
```

Note that, because these are fixed width bit vectors, even if the values of the bit vectors are the same they may not be equivalent since the number of bits used changes the type. For example, `B101 =/= X3` because the first has type `Bit 3`, which does not match the type, `Bit 4`, of the second. However, `B0101 = X3` because they both are of type `Bit 4`.

Finally, for both integer literals and bit vector literals, underscores, "_", can be used anywhere in the literal for readability, as the compiler will just ignore the underscores.

```
100000 = 100_000
0b1011 = 0b10_11 = 0b1_0_1_1
0xffff0000 = 0xffff_0000
```

2.1.2 Types and Kinds

The `Bit n` type highlights another important feature of Habit and LC, the refactored type and kind system. LC and Habit expand upon Haskell's kind system to include the standard `*` kind, also written `type`, but also includes `nat` for natural numbers, `area` for memory areas to restrict how and where memory areas can be declared, and `lab` for labels used to reference fields in `struct` and `bitdata` types. The `Bit n` constructor has kind `nat -> *`, meaning it takes a natural number and produces a new type of kind `*`. For example, the `Bit 4` stated before is of kind `*` because the number has already been specified, so it is now a first class value. This opens up a new family of types that are related and can use the same class of functions, but when used will be a distinct type that will not interact with the other types without special functions to make them equivalent. The other important type constructor with this property is the index type, `Ix n`. The index type only allows for the use of valid integers in the range [0, `n`), and is good for setting bounds. The `Ix n` type can never produce a number outside of the range [0, `n`), and trying to pass a number not in its range into a function with it's type will result in a type error. This significantly limits the potential for out of bounds errors or buffer overflows, and instead will provide a type error if something is wrong. This eliminates a large class of potential errors and is used frequently, either explicitly or predefined for certain types like `Array n a`, used to create an array with `n` components of memory layout `a`.

2.1.3 Memory Management

One of the most important features in Habit and LC that distinguishes it from Haskell and other functional languages is the ability to reserve a block of memory, initialize it, and access it later. Memory areas cannot be manipulated as first class values because they have kind `area` not `*`. Accessing memory must be done through references that make read and writes explicit. A memory area can only be initialized and reserved at the start of the program.

In addition to declaring and interacting with blocks of memory, there are special versions of type declarations that allow full control over a data type's memory layout. The first is the `struct`, similar to its C counterpart. Structs are stored as a contiguous block of memory, with each field assigned to a portion of that block of memory. Additionally, structs may specify a length such that, if the fields take up more or less space than the specified length, then the compiler will throw an error.

```
struct DLL  [ data :: Stored Unsigned
            | prev :: Stored (Ref DLL)
            | next :: Stored (Ref DLL) ]
```

Furthermore, each field within the struct can specify a memory layout, size, name, and how it should be initialized. That means whenever the struct is created the data members will be initialized using the functions set for them. Structs can also be aligned to ensure that they start at an address that is a multiple of some specific power of two. This is necessary for some baremetal applications; for example, an IA-32 page table must start at an address that is a multiple of 4KB. Finally, the named members of a struct can be individually retrieved or updated as desired.

```
initDLL   :: Unsigned -> Init DLL
initDLL v = initSelf (\self -> DLL [ prev <- initStored self
                                   | next <- initStored self
                                   | val  <- initStored v ])
```

The next special type declaration is the `bitdata` definition, which allows for the creation of types that are represented by bit vectors. Similar to `struct`, the fields in a `bitdata` type can be given names and lengths. Also similar to `struct`, a length (in bits) can be declared for a `bitdata` type. The length of the `bitdata` and its fields are specified with the `Bit n` type, and the total length of the fields must match the length of the `bitdata`.

```
bitdata Perms /3 = Perms [ read, write, grant :: Bool ]
```

Different from a struct, bitdata constructors can have multiple different types of the same length, each using their own fields. Pattern matching can be used to determine the type of the bitdata.

```
bitdata Bool /1 = False [B0] | True  [B1]
```

To actually interact with memory requires the use of the `Ref`, `Ptr`, or `Phys` types. `Ref` is the reference type, and is guaranteed to always point to a valid memory region. `Ptr` is like `Ref` but with the additional `Null` constructor, such that `Ptr` evaluates into either a `Ref` or a `Null`. `Phys` is similar to `Ref`, but uses a physical memory address not a virtual memory address.

2.1.4 Export and External

Lastly, there are several important keywords that describe how functions are compiled and if they can be seen or used by other programs. The `export` keyword makes a function visible globally, and is what lets functions be imported from one LC file or another.

The `external` keyword is used for several important features. The first use case for `external` is naming a symbol that will be declared in another part of the program. The name must be accompanied by a type to state how the compiler should treat the symbol and interact with it. This could be viewed as a Foreign Function Interface, allowing for C or assembly code to be imported for use into a Habit or LC file. The imported C or assembly code would be considered unsafe and potentially breaks some of the guarantees provided by the language, but it is sometimes necessary to do so. The `external` keyword can still help minimize and identify unsafe code. Symbols identified with `external` may also be renamed, and a constant parameter may be passed to them. [12]

```
external putchar :: Char -> Proc Unit
external wordToByte {primWordToBit 8} :: Word -> Byte
```

The other important functionality of `external` is the ability to declare an implementation for a symbol. The type of the implementation function and the type declared for the symbol do not have to be the same, but they must be equivalent. For example, a `Ref` type can be interchanged with a `Word`, because `Ref` compiles down to `Word`. When using the symbol in other parts of the program it will be treated as its declared type, while the implementation function is what will actually be called after compilation. Retyping like this once again breaks some safety guarantees, but it can be used to isolate and minimize unsafe code that other functions have to go through. For example, in the core library the `refToWord` function makes use of the external keyword to convert a `Ref` to a `Word`, which would otherwise be impossible. [12]

```
external refToWord = ptrToWordImp :: Ref a -> Word
external ptrToWord = ptrToWordImp :: Ptr a -> Word

ptrToWordImp :: Word -> Word
ptrToWordImp x = x
```

2.2 IA-32 Paging

The paging structures are defined by the IA-32 architecture, specified in Intel 64 and IA-32 Architectures Software Developer Manuals, Volume 3 [10]. Each paging structure is allocated from a 4KB page. The base address of the page directory is stored in the `%cr3` register, and first 10 bits of the 32-bit virtual address is used to index to the appropriate page directory entry (PDE). The PDEs either point to a page table, where the translation will follow Figure 1, or a a super page where the translation will follow Figure 2. If the PDE points to a super page, then the other 22 bits from the virtual address are used as the offset, with the physical address from the PDE pointing to the base of the super page. If the PDE is for a 4KB page, then the PDE will point to a page table, with the first 20 bits of the PDE holding the physical address for the base of the page table, and the next 10 bits of the virtual address used to index to the associated page table entry (PTE). Finally, the PTE will point to the base of the 4KB page being accessed, with the last 12 bits of the virtual address used as the offset for that page.

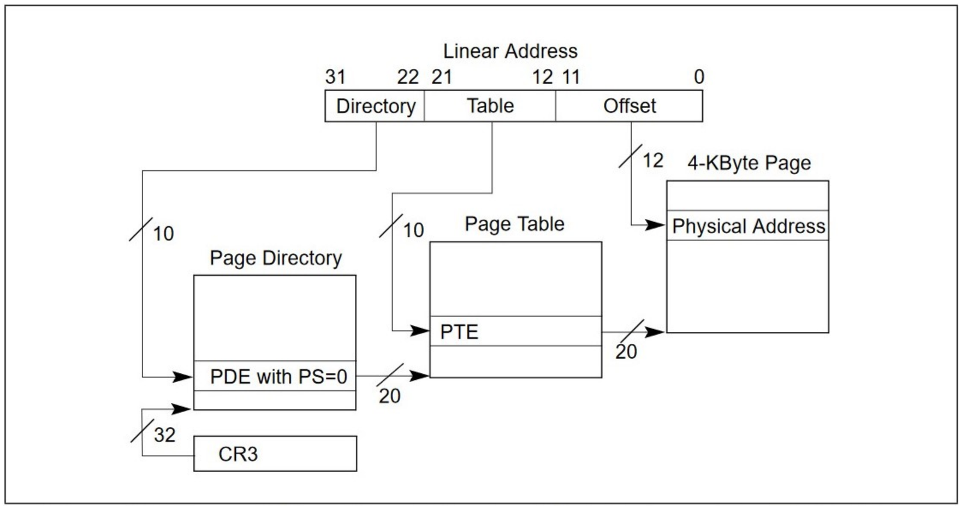


Figure 1: "Linear-Address Translation to a 4-KByte Page using 32-Bit Paging". Taken from the Intel developer manual [10]

The formats for the PDEs, PTEs, and `%cr3` are specified in Figure 3. For both PDEs and PTEs, a 0 in the last bit, or "present" bit (P), indicates that the entry should be ignored. For PDEs pointing to a super page, bits 31 to 22 specify the base of the super page, while bits 8 through 0 are the

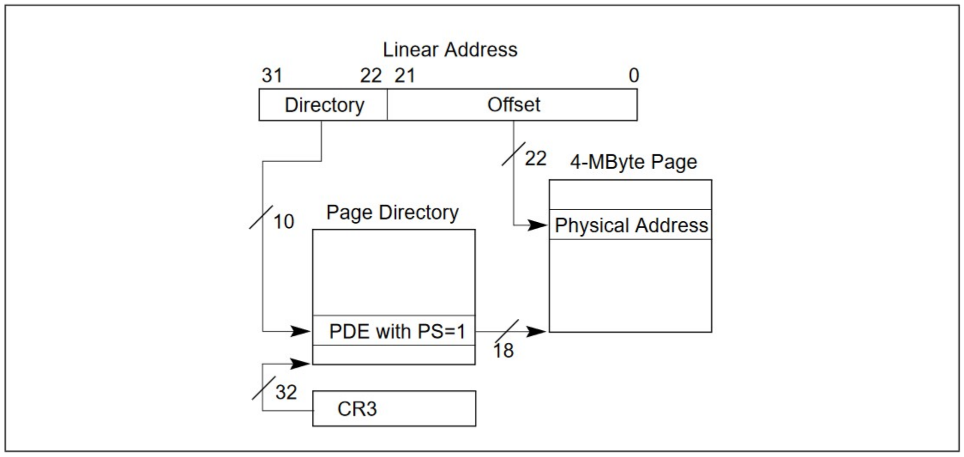
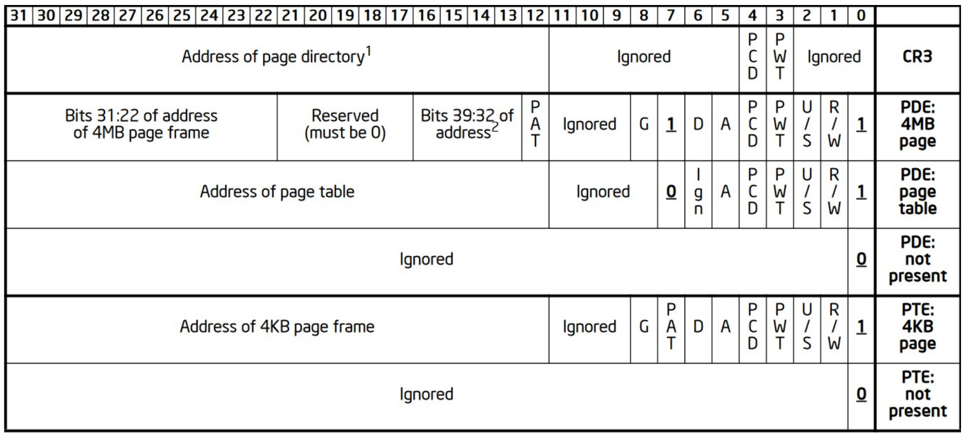


Figure 2: "Linear-Address Translation to a 4-MByte Page using 32-Bit Paging". Taken from the Intel developer manual [10]

control bits. Relevant control bits for the page table lab include bit 8 for global access (G), bit 7 set to 1 to indicate the PDE points to a super page (PS), bit 2 for user level access (U/S), bit 1 for write permission (R/W), and bit 0 (P). For PDEs pointing to a page table, bits 31 to 12 are the physical address for the start of the page table, while relevant control bits include bit 7 set to 0 (PS), bit 2 (U/S), bit 1 (R/W), and bit 0 (P). For PTEs, bits 31 to 12 point are the physical address of the page, with relevant control bits including bit 8 (G), bit 2 (U/S), bit 1 (R/W), and bit 0 (P). `%cr3` is also 32-bits, where bits 31 to 12 store the physical address of the page directory. There are only two control bits for `%cr3`, bit 4 and bit 3, neither of which were used in the page table lab.

Paging in IA-32 requires setting flags in the `%cr0` and `%cr4` registers, as well as storing a page directory's base address in `%cr3`. Setting bit 31 in `%cr0` will enable segmenting the address space in pages (PG), while setting bit 0 in `%cr0` enables protected mode (PE). Both bit 31 and bit 0 in `%cr0` need to be set to fully enable paging. To enable 4MB pages, bit 4 in the `%cr4` register must be set to 1 (PSE).



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			**CR3**
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	**PDE: 4MB page**
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**
Ignored																															0	**PDE: not present**
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**
Ignored																															0	**PTE: not present**

Figure 3: "Formats of CR3 and Paging-Structure Entries with 32-Bit Paging". Taken from the Intel developer manual [10]

3 Code Comparison

In this section, different aspects of each implementation are highlighted and discussed, to showcase some of the key differences between the C and LC code. For a more full code review, see Appendix A.

3.1 Constants

The C implementation includes several constants defined by macros, used in both the C and assembly code. These macros improved code readability and made the bit arithmetic consistent across the C and assembly code.

```
#define PAGESIZE   12
#define PAGEBYTES  (1 << PAGESIZE)
#define PAGEWORDS  (PAGEBYTES >> 2)
#define PAGEMASK   (PAGEBYTES - 1)
#define SUPERSIZE  22
#define SUPERBYTES (1 << SUPERSIZE)
```

LC does not have the functionality to share constants with the assembly code. So those macros were defined in the assembly code, with separate but equivalent constants defined in the LC code.

3.2 Bit Manipulation

In C, a PDE is created by adding the aligned physical address of a paging structure to the control bits. For the first half, an unsigned integer that represents the physical address is taken and cleared of its lower 12 bits. Then, one of the control bit related macros are added to this address, setting the lower 8 bits. But how do you verify that the correct values were used to set the PDE? First, confirm that the physical address was aligned to a 4KB boundary. For example, when creating a PDE for a page table, the page table is allocated using the `allocPage` function and casted to a `Ptab*`. All newly allocated pages are aligned to 4KB, and converting the virtual address to a physical address with `toPhys` won't affect the lower 12 bits, so the physical address should be aligned properly.

```
ptab = (struct Ptab *)allocPage();
pdir->pde[pde_index] = toPhys(ptab) + PERMS_USER_RW;
```

Secondly, to verify the macros set the correct permissions, the hexadecimal must be converted to binary to check which bits are set. Then, those bits are cross referenced with the specification to ensure the correct bits were set for the desired outcome. For example, `PERMS_USER_RW` should set the control bits such that a user can read and write to the address in that PDE. The 0x07 translates to 0000 0111, thus it should set the lowest three bits and leave the rest 0. Figure 4 is a reference diagram for this type of PDE, with the meaning of each bit explained in the Intel manual. According to the reference manual, enabling user read/write access requires setting the 2nd bit (U/S) to 1 to give users access, the 1st bit (R/W) to 1 to give users write permissions, and the 0th bit (P) to 1 to indicate that this is a valid PDE. Thus, `PERMS_USER_RW` correctly sets the bits to give a user read and write permissions.

```
#define PERMS_USER_RW 0x07
```

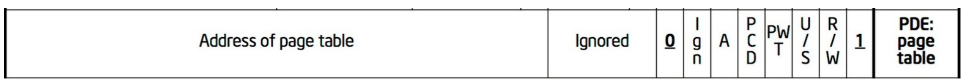


Figure 4: Specification for a PDE that points to a page table. Taken from the Intel developer manual [10]

In LC, bits for PDEs are manipulated with the `PDE` bitdata type. The

PDE type means that the correct bits will always be set and referenced, as long as the field and labels match the specification. Looking at Figure 4, the first 20 bits should be dedicated to the physical address of the page table. In the PageTablePDE constructor, ptab has type Phys PageTable, and since the PageTable type is aligned to 4KB, Phys PageTable only requires 20 bits, thus ptab is also 20 bits. Then in the PageTablePDE constructor, 4 bits are ignored, followed by the 7th bit being set to 0, the next 6 bits are various control bits represented by the PagingAttrs type, and finally the 0th bit is set to 1, which exactly matches Figure 4.

```
bitdata PDE /WordSize
    ...
    | PageTablePDE [ ptab             :: Phys PageTable
                   | unused=bit0      :: Bit 4
                   | B0
                   | attrs=readWrite  :: PagingAttrs
                   | B1 ]
    ...
```

To confirm a valid physical address is being set for a PDE, note that only a Phys PageTable type can be used for ptab. Due to the properties of the language, a Phys PageTable should always be referencing a valid physical address. So, by being able to set the ptab field at all, there is some guarantee that it will be referencing a valid memory are for a page table. Also, the 0th bit, for present, is always guaranteed to be 1 when using this type. It is also simple to confirm that the correct control bits are being set because the label for the field is used, such as us or rw, and then explicitly set to True or False. For example, the readWrite function explicitly sets the us and rw fields to True.

```
bitdata PagingAttrs /6
    = PagingAttrs [ dirty    = False     :: Bool
                  | accessed = False     :: Bool
                  | caching  = Caching[] :: Caching
                  | us                   :: Bool
                  | rw                   :: Bool  ]

readWrite:: PagingAttrs
readWrite = PagingAttrs[ us=True | rw=True ]
```

3.3 Extern and External

In C, the `extern` keyword is used to reference a global variable declared elsewhere. Adding `[]` to the end of the `extern` declaration will set the pointer with the address of the variable rather than the value of the variable. The `initdir` declaration uses this syntax, and it correctly points to the initial page directory with a `Pdir*`.

```
extern struct Pdir initdir[];
```

In LC, the `external` keyword serves a similar purpose. However, there is no way to use the address of the specified global variable, and instead it will set the value of a pointer to the value stored in the global variable. For example, `external initdir :: Ref PageDir` will set the `Ref` using the value stored at `initdir`, which would make the reference point to some other region of memory. The workaround is to declare another global variable in the assembly that holds the address of the desired symbol. For this implementation, the new global variable was named `initdir_ptr`, although it was renamed to `initdir` in the `external` declaration.

```
initdir_ptr:
    .long initdir
```

```
external initdir {initdir_ptr} :: Ref PageDir
```

3.4 Page Allocation

In C, pages are allocated by pulling from a set of usable memory that should be aligned to 4KB, converting it to a virtual address, then using that virtual address as an array to zero out the page word by word. `PAGEWORDS` is a constant that should be the number of words per page.

```
unsigned *allocPage() {
    ...
    unsigned *page = fromPhys(unsigned *, physStart);
    for (unsigned i = 0; i < PAGEWORDS; ++i) {
        page[i] = 0;
    }
    ...
    return page;
}
```

Similarly in LC, the starting address for a page is pulled from a set of usable memory, which should be aligned to 4KB, converting from a `Word` to a `Ptr`. In both the LC and C code this is an unsafe operation, and special care has to be taken to ensure that the memory being used is valid and aligned properly. The big difference is that in LC the page can be initialized using an initialization function of the desired type. For example, `allocPageDir` uses the initialization function for page directories, which effectively just zeroes out the entire page. These initialization functions are guaranteed to only set valid values for the page type, and will never go out of the page boundary.

```
allocPageDir   :: Proc (Ptr PageDir)
allocPageDir   = allocPage initPageDir

allocPage      :: Init pg -> Proc (Ptr pg)
allocPage init = do rpg <- allocRawPage
                    case rpg of
                        Null   -> return Null
                        Ref pg -> reInit pg init
                                  return rpg

external allocRawPage = allocRawPageImp :: Proc (Ptr pg)
allocRawPageImp :: Proc Word
allocRawPageImp = case<- getPageInterval physmap of
                        Just int -> return int.lo
                        Nothing  -> return 0
```

4 Discussion

LC and by extension Habit felt far more predictable than C. In C I felt the need to re-verify several times that I was using the right types, converting types in the correct places, and that the bit manipulation set the correct bits. In LC, whenever a bit vector is being changed it is clear what bit is being updated as well as what that bits purpose is, since the field names are closely related to the actual specifications labels. Additionally, when allocating pages, I felt more confident when I was able to use the initialization functions in LC since those functions are guaranteed to never go out of bounds. In the C implementation, I am still not confident that this initialization is correct, even after reverifying the math and stepping through this process several times to justify that it correctly zeroes out the entire page, but nothing else.

While LC felt more predictable than C, it only felt slightly more portable than the C code, and not as portable as its equivalent Habit code could be. The bitdata for page directory entries, and their associated functions, felt reasonably portable and easy to expand upon. Page allocation felt portable and adjustable as well, where different initialization functions could be used to set the paging structures in different ways, and more page allocation functions could easily be added if needed. Unfortunately, LC does not have type classes, which could have otherwise made for even more portable and extendable code, with potentially less boilerplate as well. One area more portable in C is that `initdir` can be explicitly referenced in the C code, which works as anticipated. However, for LC a new global variable has to be created that just stores the address of `initdir`. If there were more global variables for arrays or other objects, extra variables would be needed to reference them, thus making LC less portable when wanting to use the address of a global variable. Also, C macros can be used in both the C program and the assembly, allowing for constants to be defined that are used throughout the program. Without this, LC needs to define the constants in several places, making it less portable than C if any of the constants needed to be changed. The common theme here is that C has an easier time interacting with assembly, in a way that could easily be updated or moved to another program, which LC currently lacks.

Lastly, C is more reliable than LC. I ran into a compilation error while working on the LC implementation, preventing me from progressing further. C is an old and well tested language, used by a large number of people over many years. The same cannot be said for LC, and thus also Habit. This unreliability was anticipated, and was the driving factor for using LC over Habit. The idea was that running into a compilation error was always a possibility, but it could be potentially easier to resolve an error for LC. It's uncertain if LC compiles and optimizes in the correct and expected manner, and it won't really be possible to tell until the language is used more and feedback is subsequently given to the developers.

5 Conclusion

LC is a promising language with the potential to be great for baremetal programming. Many of the core features already aid the developer in producing more predictable code, sometimes offering a direct representation of the specification. Habit is an even more exciting language, given its extended built-in functionality, helper functions, and typeclasses to create even more portable, less redundant, code. However, the Habit and LC compiler are not currently reliable. So while there is potential, Habit and LC are not necessarily suitable for more serious projects in their current state. There are also some features that would be useful, such as additions to the external keyword to directly use a global variables address as the reference, or the ability to define constants between assembly and Habit/LC programs. To amend these problems and unknown issues, the compiler should continue to be worked on and the languages should be used in a variety of smaller projects, reporting any errors or feature gaps along the way.

Bibliography

[1] Synopsys. *2020 Open Source Security and Risk Analysis (OSSRA) Report*. 2020. URL: https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2020-ossra-report.pdf.

[2] Andy Chou et al. "An empirical study of operating systems errors". In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*. SOSP '01. New York, NY, USA: Association for Computing Machinery, Oct. 21, 2001, pp. 73–88. ISBN: 978-1-58113-389-9. DOI: 10.1145/502034.502042. URL: http://doi.org/10.1145/502034.502042.

[3] Gerwin Klein et al. "seL4: formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, Oct. 11, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.org/10.1145/1629575.1629596.

[4] MSRC Team. *A proactive approach to more secure code*. Microsoft Security Response Center. July 16, 2019. URL: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/ (visited on 08/06/2020).

[5] Rebekah Leslie and Mark P. Jones. "Operating system construction in Haskell". In: *Proceedings of the twentieth ACM symposium on Operating systems principles - SOSP '05*. the twentieth ACM symposium. Brighton, United Kingdom: ACM Press, 2005, p. 1. ISBN: 978-1-59593-079-8. DOI: 10.1145/1095810.1118589. URL: http://dl.acm.org/citation.cfm?doid=1095810.1118589.

[6] Rebekah Leslie. "A Functional Approach to Memory-Safe Operating Systems". PhD thesis. Jan. 1, 2011. DOI: 10.15760/etd.499. URL: https://pdxscholar.library.pdx.edu/open_access_etds/499.

[7] High Assurance Systems Programming Project (HASP). *The Habit Programming Language: The Revised Preliminary Report*. Tech. rep. Department of Computer Science, Portland State University, Portland, Oregon, USA, Nov. 2010. URL: https://github.com/habit-lang/language-report.

[8] Mark P. Jones, Justin Bailey, and Theodore R. Cooper. “MIL, a Monadic Intermediate Language for Implementing Functional Languages”. In: *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*. IFL 2018: 30th Symposium on Implementation and Application of Functional Languages. Lowell MA USA: ACM, Sept. 5, 2018, pp. 71–82. ISBN: 978-1-4503-7143-8. DOI: 10.1145/3310232.3310238. URL: https://dl.acm.org/doi/10.1145/3310232.3310238.

[9] Mark P. Jones. *CS 410/510 – Languages and Low-Level Programming*. URL: http://web.cecs.pdx.edu/~mpj/llp/.

[10] *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Intel. URL: https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.

[11] *habit-lang/mil-tools*. GitHub. URL: https://github.com/habit-lang/mil-tools.

[12] *zipwith/lc-baremetal*. GitHub. URL: https://github.com/zipwith/lc-baremetal.

Appendices

A Annotated Code

This appendix details the major components of the C and LC implementations, to help clarify what the code does and where each implementation differs. Both the week 4 and week 5 lab from the LLP course were implemented in C, but due to a compilation error only part of the week 4 lab was implemented in LC. So, annotations are limited to the relevant code for the week 4 lab. Both implementations are public on Github:

- C implementation: `https://github.com/dvaneson/paging`
- LC implementation: `https://github.com/dvaneson/paging-lc`

A.1 Shared Assembly

Both implementations use almost identical assembly code for setup and starting the kernel. The most important part of the shared assembly code is setting up and enabling paging before being able to use the high address space. "High address space" refers to virtual memory at 3GB and up, the kernel space, which is reserved for the kernel regardless of the program running. The program starts with no mappings and no kernel space, so everything that would normally have a virtual address starting at the kernel space or above has to be readjusted to its physical address. Also, before turning on paging, a page directory must be initialized with kernel mappings in both the low and high address space. That way everything will continue to function properly before and after moving to the high address space. Figure 5 shows how the virtual address space should be mapped, such that the upper and lower address space both map to the physical memory the instruction pointer is located in, alongside other important information. First, space for an initial page directory, `initdir`, is allocated.

```
    .globl  initdir
    .align  (1<<PAGESIZE)
initdir:
    .space  4096
```

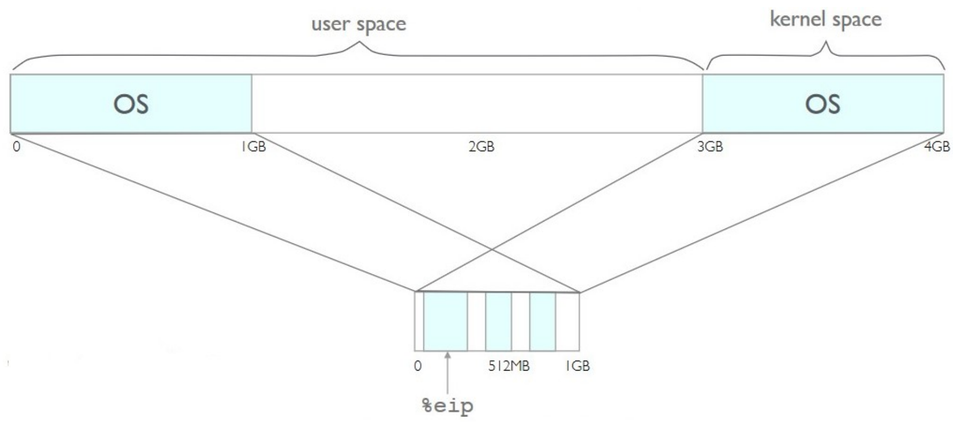


Figure 5: One to one mapping in lower and upper memory, mapping to the physical address the instruction pointer is currently in. Taken from the LLP week 4 slides [9].

Then, `initdir` is subtracted by `KERNEL_SPACE` to get the actual physical address of the intial page directory. The address is then stored in `%edi` for use in the next step, and stored in `%esi` to hold onto for later.

```
    leal    (initdir-KERNEL_SPACE), %edi
    movl    %edi, %esi
```

Next, all the memory in the page directory needs to be zeroed out. `%edi` holds the starting address of the page directory, and `%ecx` holds the number loops needed to zero the page word by word, which would be the number of words in a page, `PAGEWORDS`. In each loop the address `%edi` references is zeroed, then `%edi` is incremented by 4 to move to the start of the next word. The loop continues until `%ecx` decrements to 0.

```
    movl    $(PAGEWORDS), %ecx
    movl    $0, %eax
1:  movl    %eax, (%edi)
    addl    $4, %edi
    decl    %ecx
    jnz     1b
```

Now the initial mappings for the page directory must be made for both the low address space currently being operated in and the high address space that will be jumped to later. So, identical super page mappings are made in both the low and high address spaces. PHYSMAP is the ending address for physical memory that can be mapped to. SUPERSIZE is the bit super pages are aligned to, 22, and 2^SUPERSIZE gives the number of bytes in each super page. PHYSMAP >> SUPERSIZE is equivalent to (PHYSMAP/2^SUPERSIZE), which is how many super pages it takes to map all the memory up to PHYSMAP. PERMS_KERNELSPACE just specifies the lower 7 bits needed to set a page directory entry for super pages with only kernel read/write access. Thus, %ecx holds the counter for the loop, and %eax holds the settings for the PDEs. %edi is reset using %esi, so that it once again has the addresses starting at initdir.

```
    movl    %esi, %edi
    movl    $(PHYSMAP>>SUPERSIZE), %ecx
    movl    $(PERMS_KERNELSPACE),  %eax
```

KERNEL_SPACE is the starting address of the kernel space, 3GB. KERNEL_SPACE >> SUPERSIZE is equivalent to (KERNEL_SPACE/ 2^SUPERSIZE), which is the number of super pages needed to map up to 3GB. Alternatively, it is the starting index for the portion of the page directory that maps the kernel space, since the index starts after the 3GBs has ended. The index is multiplied by 4 since it is an array of words, or 4 byte entries. The PDEs at both low and high memory map to the same physical addresses with the same permissions, stored in %eax. %edi is incremented by 4 to move to the next PDE, because each PDE is 4 bytes. %eax is incremented by 1 << SUPERSIZE because that is the bit super pages are aligned to, and incrementing by this value will move to the start of the next super page. The loop continues until %ecx reaches 0.

```
1:  movl    %eax, (%edi)
    movl    %eax, (4*(KERNEL_SPACE>>SUPERSIZE))(%edi)
    addl    $4, %edi
    addl    $(1<<SUPERSIZE), %eax
    decl    %ecx
    jnz     1b
```

After initdir is initialized, paging is enabled by setting the correct values in the control registers. First, the page directory address in %cr3 is set to initdir. Then, bit 4 of %cr4 is set to 1 to allow for super pages. Lastly, set bit 0 and bit 31 in `%cr0` are set to 1 to turn on paging.

```
    movl    %esi, %cr3

    mov     %cr4, %eax
    orl     $(1<<4), %eax
    movl    %eax, %cr4

    movl    %cr0, %eax
    orl     $((1<<31)|(1<<0)), %eax
    movl    %eax, %cr0
```

Finally, the program jumps to the high address space before calling the kernel. The address of `high` is stored in `%eax`, then `%eax` is used to jump. `jmp $high` would not have jumped to the high address space because it would only jump locally, staying in the low address space but moving down one line. If `jmp $high` did jump to the high address space, then the previous loops would have also jumped into the high address space.

```
    movl    $high, %eax
    jmp     *%eax
high:
    ...
```

A.2 C

A.2.1 Macros

For the C implementation various code was given up front, such as switching between virtual and physical memory. The self made distinction here is that physical addresses are typically represented by an unsigned integer, whereas a virtual address is represented by a pointer, adjusted to be within the kernel space, 3GB and above. The following macros are used to convert between the two types of addresses.

- `fromPhys` takes the physical address, casts it to an unsigned integer just in case, adds `KERNEL_SPACE` to it, then casts it to pointer type, `t`.
- `toPhys` takes a pointer, `ptr`, casts it to an unsigned integer, then subtracts `KERNEL_SPACE` from it.

```
#define fromPhys(t, addr) ((t)((unsigned)(addr)+KERNEL_SPACE))
#define toPhys(ptr)       ((unsigned)(ptr)-KERNEL_SPACE)
```

There are also several macros to clear the bits for a given variable,

- `maskTo` takes a variable, `e`, and an integer, `a`, and clears the bits from bit position `a` and up, leaving only the lower bits left. To achieve this, first 1 is shifted to bit position `a`, and after subtracting 1 all bits below position `a` will be set to 1. Taking the bitwise and of this value and `e` results in bits at position `a` and above being cleared for `e`.

- `alignTo` takes a variable, `e`, and an integer, `a`, and clears the bits below bit position `a`. To achieve this, `e` is shifted right `a` times, leaving only bits `a` and above untouched, then `e` is left shifted `a` times, clearing all bits below `a`.

```
#define maskTo(e, a)  ((e) & ((1 << (a)) - 1))
#define alignTo(e, a) (((e) >> (a)) << (a))
```

Pages must be aligned to 4KB, such that the top 20 bits of a physical address correspond to the start of the page. Several macros are provided to find the start and end of a page, as well as moving between pages. These macros should work as shown in Figure 6

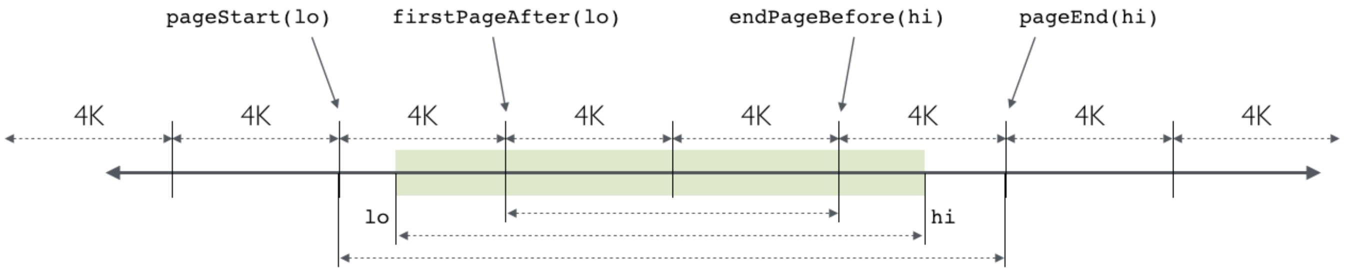


Figure 6: Page boundaries and the effect the macros should have. Taken from the LLP week 4 lab [9].

- `PAGEMASK` is 0xfff

- `pageStart` should return the address of the first byte in the page that contains address `x`. To achieve this, `PAGEMASK` is negated to produce the value 0xffff_f000, which is then used to bitwise and `x`, clearing its lower 12 bits, and aligning it to the start of the page.

- `pageEnd` should return the address of the last byte in the page that contains address `x`. To achieve this, `x` is bitwise or'd with `PAGEMASK`, producing the end of the page for address `x`.
- `pageNext` should return the address of the first byte in the page that comes immediately after the page containing `x`. To achieve this, the end of the page is found with `pageEnd`, then adding one to the end of the page will result in the start of the next page.
- `firstPageAfter` should return the address of the first page whose starting address is greater than or equal to `x`. Different from `pageNext` since it can return `x` if it is at the start of the page. To achieve this, `PAGEMASK` is added to `x`, then the start of the page for the new address is found using `pageStart`. If `x` was the start of a page, then adding `PAGEMASK` to it will bring it to the end of the page, and `pageStart` will return to the original address, `x`. Otherwise, the start of the next page will be returned.
- `endPageBefore` should return the end address of the first page whose end address is less than or equal to x. To achieve this, `PAGEMASK` is subtracted from `x`, then the end of the page for the new address is found using `pageEnd`. If `x` was the end of a page, then subtracting `PAGEMASK` from it will bring it to the start of the page, and `pageEnd` will return to the original address, `x`. Otherwise, the end of the previous page will be returned.

```
#define pageStart(x)      ((x) & ~(PAGEMASK))
#define pageEnd(x)        ((x) | PAGEMASK)
#define pageNext(x)       (pageEnd(x) + 1)
#define firstPageAfter(x) (pageStart((x) + PAGEMASK))
#define endPageBefore(x)  (pageEnd((x) - PAGEMASK))
```

A.2.2 Memory Map

The page table requires page sized regions to create the various paging structures. The memory for these pages is found by looking through the memory map, `mmap`, located in the bootdata. The data for each `mmap` is stored in a contiguous block for the starting and ending address. For a region to be valid, the following conditions must be met.

- Start address aligned to 4KB (0x000 offset)
- End address aligned to 4KB-1 (0xfff offset)
- Start is less than end
- Start comes after `KERNEL_LOAD`, where the kernel is loaded (1MB)
- End must come before `PHYSMAP` (32MB)
- Of the set of valid regions, the largest region is selected to be used for page allocation, denoted by `physStart` and `physEnd`.

```
physStart = 0;
physEnd = 0;
for (i = 0; i < mmap[0]; i++) {
    start = firstPageAfter(mmap[2 * i + 1]);
    end = endPageBefore(mmap[2 * i + 2]);
    if (start >= KERNEL_LOAD && end < PHYSMAP && start < end)
    {
        if (physEnd - physStart < end - start) {
            physStart = start;
            physEnd = end;
        }
    }
}
```

After looping through `mmap`, if there were no valid regions the program terminates and reports a fatal error. If there is a valid region, then the last step is to look for conflicts in the [`physStart`, `physEnd`] region for memory already in use, as specified by `headers`. `headers` is stored similarly to `mmap`, with each header being a contiguous block of the region start, end, and the entry number. If there's any overlap, that region is removed from [`physStart`, `physEnd`].

```
if (physEnd <= physStart) {
    fatal("Could not find a valid region in memory map.");
}
```

```
for (i = 0; i < hdrs[0]; i++) {
    start = hdrs[3 * i + 1];
    end = hdrs[3 * i + 2];
    if (end < physEnd && end > physStart)
        physStart = firstPageAfter(end + 1);
    if (start > physStart && start < physEnd)
        physEnd = endPageBefore(start - 1);
}
```

A.2.3 Allocating Pages

Next, the memory region is used to allocate pages. Allocating a page requires setting aside memory starting at `physStart` and ending right before the next page boundary, `pageEnd(physStart)`. So the first checks are just to make sure there is enough memory between `physStart` and `physEnd` for allocation, and if there isn't the program terminates and reports a fatal error. After determining the region [`physStart`, `pageEnd(physStart)`] is usable, the region needs to be zeroed out. To do this, the `physStart` physical address is converted to a virtual address, represented by an unsigned pointer, and this pointer is then used to access this memory region. `PAGEWORDS` is the number of words in a page, which is used as the stopping point because each `page[i] = 0` zeroes out a word, and the next index will start at the next word. After finishing the for loop, `physStart` has to be updated to the start of the next page, and the pointer to the newly zeroed page is returned.

```
unsigned *allocPage() {
    if (physEnd < pageEnd(physStart)) {
        fatal("Could not allocate a page");
    }

    unsigned *page = fromPhys(unsigned *, physStart);
    for (unsigned i = 0; i < PAGEWORDS; ++i) {
        page[i] = 0;
    }
    physStart = pageNext(physStart);
    return page;
}
```

Now that pages can be initialized and referenced, various paging structures can be created. The first paging structure to initialize is the page directory. Each page directory needs to have mappings in the kernel space, which are the same mappings as the assembly that added super page mappings in the kernel space for the first `PHYSMAP` bytes of memory. First, a page is allocated for the page directory, casted from an `unsigned*` to a `Pdir*`. Then the number super pages needed to map the usable region of physical addresses is stored in `kern_entries`, using the same bit shifting as the assembly code. The starting index for the kernel space is stored in `kernel_addr`, using similar bit shifting to the assembly code, but without the need to multiply by 4. Finally, the physical address of the super pages and the control bits for each super page mapping must be set. The physical address for the super page mappings will be at `i * SUPERBYTES`, because `SUPERBYTES` is the number of bytes per super page, 2^{22}, and multiplying by `i` gives the starting address for the ith super page. The control bits are added to the physical address because only the 31st through 22nd bits are set, so adding `PERMS_KERNELSPACE` to it will just set the lower bits to the value of `PERMS_KERNELSPACE`.

```
struct Pdir *allocPdir() {
    struct Pdir *pdir = (struct Pdir*)allocPage();
    unsigned kern_entries = (PHYSMAP >> SUPERSIZE);
    unsigned kern_addr = (KERNEL_SPACE >> SUPERSIZE);

    for (unsigned i = 0; i < kern_entries; ++i) {
        pdir->pde[i + kern_addr] = (i * SUPERBYTES) +
                                    PERMS_KERNELSPACE;
    }
    return pdir;
}
```

Allocating the page tables only requires calling `allocPage` and casting it to a `Ptab*`.

```
struct Ptab *allocPtab() {
    return (struct Ptab*)allocPage();
}
```

A.2.4 Mapping Pages

Finally, a 4KB page can be mapped in the user space of the page directory. To map the page requires the page directory, the virtual address to associate with the page, and the physical address of the page. Despite being a virtual address, `virt` is an unsigned integer. Both `virt` and `phys` are aligned to 4KB, clearing their offset.

```
void mapPage(struct Pdir *pdir, unsigned virt, unsigned phys)
{
    virt = alignTo(virt, PAGESIZE);
    phys = alignTo(phys, PAGESIZE);
```

The PDE index is stored in bits 31 to 22 of the virtual address. So, `pde_index` is set by shifting `virt` until all that is left is bits 22 and up. The PTE index is stored from bits 21 to 12, so to single out those bits the upper 12 bits must be cleared with `maskTo`, then the result from that operation is shifted 12 bits and stored in `pte_index`.

```
    unsigned pde_index = (virt >> SUPERSIZE);
    unsigned pte_index = maskTo(virt, SUPERSIZE) >> PAGESIZE;
```

Then `pde_index` is checked to ensure it is within bounds. If it isn't, panic and report a fatal error. Otherwise, the PDE is found and stored in `pde`.

```
    if (pde_index >= PAGEWORDS) {
        fatal("PDE index out of bounds");
    }
    unsigned pde = pdir->pde[pde_index];
```

Then, `pde` is checked for the present bit. If the present bit is set to 1, then `pde & 1` will return true. Next, `pde` is checked to ensure a super page is not already mapped at that address, and if a super page is already mapped for `pde` the process terminates and reports a fatal error. Otherwise, the lower 12 bits of `pde` are cleared, leaving only the physical address to the base of the page table. The page table address is subsequently converted to a virtual address represented by a `Ptab*`. One last check before mapping the PTE ensures that the PTE is not already mapped to another page, and if it is then the process terminates.

```
    struct Ptab *ptab = 0;
    if (pde & 1) {
        if (pde & PERMS_SUPERPAGE) {
            fatal("PDE maps to a superpage");
        }

        ptab = fromPhys(struct Ptab*, alignTo(pde, PAGESIZE));
        if (ptab->pte[pte_index] & 1) {
            fatal("PTE already mapped to a physical address");
        }
    }
```

If `pde` was unmapped, then a new page table is allocated, with `pde` set using the physical address of the new page table as well as user read/write permissions, `PERMS_USER_RW`. Finally, after ensuring a valid PTE, the PTEs upper 20 bits are set by `phys` and the lower 12 control bits are set with user read/write permissions, `PERMS_USER_RW`.This addition works because at the beginning of the function the lower 12 bits of `phys` were cleared.

```
    else {
        ptab = (struct Ptab*)allocPage();
        pdir->pde[pde_index] = toPhys(ptab) + PERMS_USER_RW;
    }

    ptab->pte[pte_index] = phys + PERMS_USER_RW;
}
```

A.3 LC

A.3.1 Types

Since this is a functional programming language, one of the first steps is defining types. texttttPageSize and `SuperPageSize` types defined by numeric literals, namely 4KB and 4MB respectively. Then, the `Page` and `SuperPage` structs use these types to define their length, alignment, and the size of the array inside each struct.

```
type PageSize      = 4K
type SuperPageSize = 4M

struct SuperPage /SuperPageSize
```

```
    [ bytes :: Array SuperPageSize (Stored Byte) ]
    aligned SuperPageSize

struct Page /PageSize
    [ bytes :: Array PageSize (Stored Byte) ]
    aligned PageSize
```

The PagingAttr type defines the common control bits shared between the various paging types. The Caching type likewise defines the bits used for caching. Each field of the bitdata is given a type, usually Bool which has type Bit 1 with constructors True and False. Several of the bitdata fields are given a default initialization value as well. For example, the dirty and accessed bits are set to False by default, and the caching field is initialized with the default Caching object, denoted Caching[], which in turn will be initialized with pcd and pwt set to False.

```
bitdata PagingAttrs /6
    = PagingAttrs [ dirty    = False     :: Bool
                  | accessed = False     :: Bool
                  | caching  = Caching[] :: Caching
                  | us                   :: Bool
                  | rw                   :: Bool  ]

bitdata Caching /2
    = Caching [ pcd=False, pwt=False :: Bool ]
```

There are several useful functions that return a PagingAttr bitdata with specific fields set. For example, readWrite will return a PagingAttr that has the us and rw field set to True, meaning that a user can access the paging structure, and the user has permission to read and write to the paging object. Similarly, the kernelOnly function returns a PagingAttr with us set to False and rw set to True, meaning only the kernel will have read and write permissions for the paging structure.

```
readWrite :: PagingAttrs
readWrite = PagingAttrs[us=True | rw=True]

kernelOnly :: PagingAttrs
kernelOnly = PagingAttrs[us=False | rw=True]
```

Page table entries use the PTE bitdata type. The PTE bitdata has a length of WordSize, where the value of WordSize depends on the architecture. In IA-32 words are 32 bits, so WordSize equals 32. B0 and B1 mean that the bit in that position will always be 0 or 1 respectively, but the bit doesn't have a name associated with it. Thus a PTE can either be an UnmappedPTE if the entries present bit is set to 0, or a MappedPTE if the entry points to a page. bit0 is a function that returns a zeroed out bit vector of the appropriate length. Finally, references through Phys or Ref are usually represented through a Word, which are 32 bits. But, since Page is aligned to 4KB, the Phys Page type can be referenced using 20 bits instead of 32, and thus the page field is 20 bits.

```
bitdata PTE /WordSize
    = UnmappedPTE  [ unused=bit0  :: Bit 31
                   | B0 ]
    | MappedPTE    [ page         :: Phys Page
                   | unused=bit0  :: Bit 3
                   | global=False :: Bool
                   | pat=False    :: Bool
                   | attrs        :: PagingAttrs
                   | B1 ]
```

PageTable is a struct with length and alignment PageSize, or 4KB. PageTable has one field, pte, an array of 1KB PTEs. The initialization function for PageTable simply sets each PTE in the array to UnmappedPTE, which effectively zeroes out the entire array.

```
struct PageTable /PageSize
    [ ptes :: Array 1K (Stored PTE) ]
    aligned PageSize

initPageTable :: Init PageTable
initPageTable = PageTable [ ptes <- initArray (\ix ->
                                    initStored UnmappedPTE[])]
```

Page directory entries use the PDE bitdata type. A PDE can either be UnmappedPDE for unmapped entries with the present bit set to 0, PageTablePDE for entries that point to a page table, or SuperPagePDE for entries that point to a super page. In PageTablePDE the previously mentioned readWrite function is used to initialize the value of attrs.

Since PageTable is aligned to 4KB, a Phys PageTable only requires 20 bits, and thus ptab is 20 bits. Similar for Phys SuperPage, except that SuperPage is aligned to 4MB, and thus super in SuperPagePDE is 10 bits.

```
bitdata PDE /WordSize
    = UnmappedPDE  [ unused=bit0     :: Bit 31
                   | B0 ]
    | PageTablePDE [ ptab            :: Phys PageTable
                   | unused=bit0     :: Bit 4
                   | B0
                   | attrs=readWrite :: PagingAttrs
                   | B1 ]
    | SuperPagePDE [ super           :: Phys SuperPage
                   | unused=bit0     :: Bit 13
                   | global=False    :: Bool
                   | B1
                   | attrs           :: PagingAttrs
                   | B1 ]
```

The KPDE type is similar to PDE, but intended for page directories in the kernel space. The KPDE bitdata has two constructors, UnmappedKPDE for unmapped pages and SuperPageKPDE for super pages. Instead of a Phys SuperPage field, an index is used. This is to further separate KPDE from the user level PDE, limiting the overlapping types. The Ix type is set with KernelSuperPages1, or 255, which is the number of super pages that could be mapped in the kernel space minus one to leave space for a buffer page. Since only 8 bits are needed to store a number between 0 and 254, the beginning is padded with two 0 bits to make the total length 32 bits.

```
bitdata KPDE /WordSize
    = UnmappedKPDE  [ unused=bit0 :: Bit 31
                    | B0 ]
    | SuperPageKPDE [ B00
                    | ix          :: Ix KernelSuperPages1
                    | unused=bit0 :: Bit 13
                    | global=True :: Bool
                    | B1
                    | attrs=kernelOnly :: PagingAttrs
                    | B1 ]
```

PageDir, unlike the PageTable, has three fields. The first is for user level mappings, defined by the field pdes, which is an array of UserSuperPages, or 756, PDEs. The second is for kernel level mappings, defined by the field kpdes, which is an array of 255 KPDEs. The last field is bufferPDE, which is a single PDE used for a buffer page. PageDir has length and alignment PageSize, or 4KB. The initialization function for PageDir, initPageDir, goes through each field and initializes each index of the array to an unmapped type, effectively zeroing out the entire struct.

```
struct PageDir /PageSize
    [ pdes      :: Array UserSuperPages    (Stored PDE)
    | kpdes     :: Array KernelSuperPages1 (Stored KPDE)
    | bufferPDE :: Stored PDE ]
    aligned PageSize

initPageDir :: Init PageDir
initPageDir
    = PageDir [ pdes  <- initArray (\ix ->
                                    initStored UnmappedPDE[])
              | kpdes <- initArray (\ix ->
                                    initStored UnmappedKPDE[])
              | bufferPDE <- initStored bufferPtabPDE ]
```

A.3.2 External Objects

After the paging types are setup, the kernel can reference and use the page directory initialized by the assembly, initdir. In order to have a proper reference to initdir an additional variable had to be declared in the assembly, initdir_ptr, which just holds the address of initdir.

```
initdir_ptr:
    .long initdir
```

Then, initdir_ptr is used with external to reference the initial page directory, by casting it to type Ref PageDir and renaming it initdir. Once cast to Ref PageDir, the types of each member of the pdes array will be determined based on what the last bit and 7th bit are set to. If the last bit is set to 0, the member will be typed UnmappedPDE, if the last bit is set to 1 and 7th bit is set to 0, it will typed PageTablePDE, and if the last bit is set to 1 and the 7th bit is set to 1, it will typed `SuperPagePDE`. Similar is true for the `kpdes` and `bufferPDE`.

```
external initdir {initdir_ptr} :: Ref PageDir
```

To create the paging objects, available memory has to be set aside. Once more the usable physical addresses will be found through the memory map and headers, stored in the bootdata. To acquire the bootdata, `external` is used with the address 0x1000, typed to `Ref MimgBootData`. This creates a reference to memory address 0x1000, that will be accessed as a `MimgBootData` type. The specifics of `MimgBootData` can be found in mimg.llc.

```
external bootdata = 0x1000 :: Ref MimgBootData
```

A.3.3 Memory Map

Now `bootdata` can be passed to `initPhysMap`, exported from the paging.llc file. The paging.llc file also contains an area declaration for `physmap`, an `IntervalSet` for the physical memory available for page allocation. For more information on the `IntervalSet` type, refer to intervals.llc. The `IntervalSet` type defines a series of disjoint intervals, where overlapping intervals are merged into one larger interval.

```
area physmap <- IntervalSet[] :: Ref IntervalSet
```

There are several helper functions to help with the partitioning and inserting of intervals into a given `IntervalSet`. The first is `insertInterval`, which simply inserts the interval into the set, looking first to see if the interval can be merged with an existing interval, or added as a disjoint interval otherwise. This function is used in conjunction with the `mmap` portion of `bootdata` to create an `IntervalSet` out of the memory map.

```
insertMMaps :: MimgMMapCursor -> Proc Unit
insertMMaps mmapCursor
    = forallDo nextMimgMMap insertMMap mmapCursor
    where
        insertMMap :: Ref MimgMMap -> Proc Unit
        insertMMap mmap
```

```
            = do start <- get mmap.start
                 end <- get mmap.end
                 insertInterval physmap Interval[ hi=end
                                                | lo=start ]
```

The `reserveInterval` function scans the `IntervalSet` for conflicting intervals, removing them or partitioning them if the specified interval overlaps with any interval already in the set. The `headers` in `bootdata` are regions of memory already in use, so `headers` are iterated over with `reserveInterval` to remove any intervals already in use.

```
reserveHeaders :: MimgHeaderCursor -> Proc Unit
reserveHeaders headerCursor
    = forallDo nextMimgHeader reserveHeader headerCursor
    where
        reserveHeader :: Ref MimgHeader -> Proc Unit
        reserveHeader header
            = do start <- get header.start
                 end <- get header.end
                 reserveInterval physmap Interval[ hi=end
                                                 | lo=start ]
```

Several other parts of `physmap` need to be reserved as well, such as memory below 1MB for the kernel, and memory above 32MB since that's where the usable physical memory section ends. `valdiateIntervals` ensures that the intervals start aligned to 4KB and end at 4KB-1, to match page boundaries.

```
initPhysMap :: Ref MimgBootData -> Proc Unit
initPhysMap bootdata
    = do mmapCursor <- mimgMMap bootdata
         insertMMaps mmapCursor
         reserveInterval physmap kernInterval
         reserveInterval physmap physmapInterval
         headerCursor <- mimgHeaders bootdata
         reserveHeaders headerCursor
         validateIntervals physmap
    where
        kernInterval
            = Interval[ hi = 1M-1 | lo = 0 ]
        physmapInterval
            = Interval[ hi = 0xffff_ffff | lo = 32M ]
```

A.3.4 Allocating Pages

Finally, paging objects need to be allocated for the page table. This allocation is tricky because memory is typically initialized and typed at the start of the program, to ensure that all memory references are valid and typed correctly. Additionally, the intervals use type `Word`, but typically you are not allowed to convert from a `Word` to a `Ref` because the `Word` could have been any value, and thus there is no guarantee that after converting it will be a valid reference. The `external` keyword can be used to ignore both of these rules. First, `allocRawPageImp` is declared, which will try to pull a 4KB interval from `physmap`, and if it does then the low address of the interval is returned to be used as a page, otherwise 0 is returned. Then, the `allocRawPage` function is defined with `external` such that it has type `Proc (Ptr pg)`, with its implementation set to `allocRawPageImp`. This means that `allocRawPage` will be used everywhere else in the program as type `Proc (Ptr pg)`, while the underlying implementation has type `Proc Word`, thus converting from a `Word` to a `Ptr`. If `allocRawPageImp` returns a 0, it will be converted to a `Null`, and if it returns anything else it will be converted to a `Ref`. However, this conversion is unsafe and `allocRawPage` breaks the type and reference guarantees. Thus, care should be taken to limit the unsafe code and manually verify its correctness.

```
external allocRawPage = allocRawPageImp :: Proc (Ptr pg)
allocRawPageImp :: Proc Word
allocRawPageImp = case<- getPageInterval physmap of
                        Just int -> return int.lo
                        Nothing  -> return 0
```

`allcoPage` takes an initialization function of the same type as the returned `Ptr`'s inner type. `allocPageRaw` is called to retrieve the raw page, and if the `Ptr` to this page is `Null`, `Null` is once again returned. Otherwise, the initialization function is used with `reInit` to initialize the memory area. Then, the `Ptr` for this initialized area is returned.

```
allocPage      :: Init pg -> Proc (Ptr pg)
allocPage init = do rpg <- allocRawPage
```

```
                    case rpg of
                        Null   -> return Null
                        Ref pg -> reInit pg init
                                  return rpg
```

Finally, `allocPagedir` and `allocPageTable` call `allocPage` with their respective initialization functions. Both functions return a `Ptr` which will either be `Null` or a reference to a freshly initialized memory area of the corresponding type. These are the only two allocation related functions exported and usable outside of this file.

```
export allocPageDir, allocPageTable
allocPageDir   :: Proc (Ptr PageDir)
allocPageDir   = allocPage initPageDir

allocPageTable :: Proc (Ptr PageTable)
allocPageTable = allocPage initPageTable
```