1 Open Questions

How much should we require the read-only-ness of executables?
How do we handle synchronizing objects across multiple machines in a way that is scalable?
What kind of default naming service should we have? What kind of default paging service should we have?

The primary requirement for pointers is A pointer that is stored within an object is stored in local-id:offset form from the perspective of the object that it is stored in.
An additional requirement for a flat-address-space machine is A pointer that is stored in a register MUST be in a form usable by the CPU.

2 Application Program Perspective - take 3

2.1 A model: no modification of programs
An executable image becomes very simple. It is going to have, in its foreign object table, a reference to data that it needs to access during code execution. Code and data is something that (in most cases) we don’t want to have changes to one running instance propagate to another running instance. We require that these objects be read-only (this is an open question). If you want to update a program, you create a new object.

Since data that a program needs to access are provided as objects in the FOT (foreign object table), we can specify that those are copy-on-write if we choose, so that running instances don’t have their data changed out from under them. Of course, we may want some objects to be shared, so we don’t set copy-on-write for those.

Other objects that an executable image may reference are shared libraries, which are of course, just objects. This makes an external call very simple: call obj-id:offset.

If we take the global ID of an object to be literally global, then it easily follows that most objects on a system will be cached local copies of global objects. If I have an executable image that references a library with some ID $x$, then I only need $x$ to be in my local memory if I actually run that program. If the program sits around for years and is never run, then I can probably delete that object in order to reclaim some space. I can retrieve it again from someone who has it later if I need it. If the writer of library $x$ wants to update the library, they don’t release a modified version of $x$, they release a new library called $y$.

FOT entries can point not only to global IDs, but also to names that get resolved at load-time of that object. This means that a shared library entry would point to the “most recent revision of version $x$”.

2.2 Transient Objects

Another thing we need in a program is a zero-initialized section. This section needs to be zero at load-time, but is also important for program state. When an executable image is unloaded, it can be discarded. A more general case (and one that fits with TLS objects): when the last reference to this object is removed, it is deleted. For this kind of thing, we can have an entry in the FOT (say the executable image refers to zero-initialized memory as local object ID 4) that
says “entry 4: unallocated, transient, zero-initialize”, or something along those lines. Another possibility is to have a “zero-object” marked as copy-on-write which has an unspecified size.

3 x86

An example:

```c
int a;
void foo(int *ptr) { }
void main(void) { int *x = &a; foo(x); }
```

The generated (and annotated) assembly code (with no optimizations) is:

```assembly
1 <foo>
2 void foo(int *ptr) {
3 push rbp
4 mov rbp, rsp
5 mov [rbp-0x8], rdi
6 nop
7 pop rbp
8 ret
9 }
10 <main>
11 void main(void) {
12 push rbp
13 mov rbp, rsp
14 sub rsp, 0x10
15 int *x = &a;
16 mov [rbp-0x8], 0x6008e4
17 foo(x);
18 mov rax, [rbp-0x8]
19 mov rdi, rax
20 call <foo>
21 nop
22 leave
23 ret
24 }
```

Obviously, for these instructions to work, we need the registers to hold value that the CPU can actually operate on. For a flat address space, the stack pointer will be of the form `addrspace-local-id:offset`. Line 16 would change to be `mov [rbp-0x8], rdata:0x6008e4`.

4 Application program perspective

Some more ramblings about applications

(I’m going to write in C. Many of the functions that I’ll write should probably be elided by the compiler automatically in the case where it CAN determine what to do.)

Let’s take a look at a very simple application of this global addressing scheme: client/server communication. Let’s say the server does something to ‘create’ or somehow acquire an object:

```c
object *obj = object_map("cthulhu");
```

Now, this server is going to want to react to changes to the object from outside sources. Let’s assume that there is a way to do that:

```c
object_notify_handler(obj, OBJECT_MODIFIED, handler);
```

Now, the idea is that when the object changed, the handler gets called, and the server can do something with the data inside. Preventing race conditions is not my problem. We can also do things like this with some kind of “wait for object to be ready to be read” (*cough* `select` *cough*), or just some handy library functions for reading sequential data.
The client gets the object in much the same way. It can then write some data (like a string, http request, whatever), and run something like `object notify(obj)` to indicate that it changed an object. 

Note that because we can have the client and server refer to the same object, we’ve actually just gotten socket-like shared memory across multiple computers. It may not be fast if you do that, but sharing data between computers wasn’t fast anyway. And we’ve gotten that functionality without calling socket, bind, listen, etc.

5  Objects

Something about object sharing: If an image says that it links against a certain object (with some global ID), and that image is then moved to a different computer, we’d need to have some way for that object that the image links with to get to the new computer that we’re running this image on. This leads to an interesting implication: you don’t actually have to install libraries on a computer, you can just get them from somewhere when you load a program, and it can stick around on your computer as long as it doesn’t get evicted because of lack of space...As long as it exists somewhere in the world, you can get it.

6  Name Service

Example: read some data from a file on a remote server. IF WE KNOW the global ID of the object we want, then we can just map it and use it. But it’s more likely that we want to refer to it by a more useful name. So perhaps we can say something like `guid = name_resolve("com.example.remote : /some/path/to/file");` if we want to (or whatever naming conventions... “/com/example/remote/some/path/to/file”)

Above, we said that we can have shared libraries. So, this `name_resolve` function would be exposed by some library that provides a convenient way for the program to interact with the name service. Now, the name service may have just be some library that does stuff when the functions are called, or it’s a seperate running program that gets interacted with, I don’t know.

We can allow multiple naming services to exists, since they just map name to guid. However, the kernel needs to be able to handle at least one, because FOT entries can have names in them. This can be a very limited name resolution should the kernel decide to do that.

7  POSIX Wrapper

8  Beyond POSIX

Allow an executable to specify its own pagers.

9  Security

merkle trees!

Here are a bunch of ramblings about a flat address space implementation and some stuff on arm vs x86 that a tossed together before a midterm....
10 Implementation in a flat address space

Because current architectures are implemented via a flat address space, we need to describe a way to map the desired system to one that can be implemented on today’s hardware. ... The goals of this should be to specify a system such that application software does not need to know what system it is running on (for the sake of portability). However, the operating system kernel and userspace library will need to know and adjust its operation to match.

A virtual address space contains a set of objects that are currently mapped in. This is done on-demand via a page-fault handler.

On object exists in virtual memory, when mapped, in a given location that is determined when it is loaded. We can specify a maximum object size of, say, 4GB. Then bits [31:0] are offsets inside the object and bits [47:32] refer to the location in the address space where it is mapped.

There are three types of object IDs in this design (which is just one possible design). Global IDs are globally unique object identifiers, object-local IDs are IDs used by pointers within an object (object-local ID of 0 refers to its own object), and address-space-local IDs are locations where objects are currently mapped into the virtual address space.

The local-ID is translated per-object. This means that each object needs to have a table associated with it to map local-ID to global-ID.

The third type of ID is necessary in the case where two different objects that are mapped into the same address space refer to local-ID 1. These may be different objects, and so would be mapped into different locations in the virtual address space. Another case where this is needed is when an object is mapped into two different address spaces, and refers to local-ID 1. The object that it is referring to may be mapped into different locations inside each address space.

Thus there needs to exist a mechanism to get the address-space-local ID for a given object. Pointers to objects occur inside an object by referring to the object-local ID and being translated using a table present for each object. Another translation table can exist per address space, which translates a global ID into an address-space-local ID. In the case described above, the address-space-local ID is bits [47:32] of an address. This is 16 bits, allowing each address space to have 65K objects mapped in at any one time.

11 Arm64 vs. x86

For currently existing hardware, we are looking at both x86_64 and ARM64 (aarch64/armv8). Each has a distinct set of advantages. The main advantage of x86_64 is availability, since we already have numerous such machines. Arm64 is more limited because it’s much newer, so it doesn’t yet have many choices for development boards (though there are some). Testing on x86_64 would be easier because we can use virtualization on our x86_64 workstations. Arm64 can only be emulated on these machines.

However, arm64 supports tagged pointers. This allows software to use the highest 8 bits of a virtual address to store information, whereas on x86_64 the upper 8 bits are not usable.

On x86_64, a virtual address is determined by bits 0 through 48, with all higher bits sign-extended from bit 48. This gives two ranges of address, 0000000000000000h through 0000FFFFFFFFFFFFFFh, and FFFF000000000000h through FFFFFFFFFFFFFFFFFh. The upper 16 bits must be either all 0 or all 1.

On arm64, the same is true, except that tagged pointers may use the upper 8 bits, and these bits are ignored during virtual address translation.

This does not extend the size of the address space, as a pointer that has the upper 8 bits set and another that doesn’t will translate to the same physical address. But I’m sure we can make use of those 8 bits somehow. One possible use would be local IDs. We can have more than 65K objects that an object references, you just may not be able to load in all of them into the address space at the same time. So we’re back to swapping.

1. In terms of ability, both are similar.
2. ARM has cheaper context switching (don’t need to change the kernel-mode page tables).
3. Both can access up to 48bits of physical ram.
4. ARM64 is much simpler, which makes it easier to quickly get started.

5. ARM64 is harder to test (or test well). Emulation or virtualization are great way to test operating system kernels. For x86_64 we can use virtualization via virtualbox/kvm/etc, but for ARM64 we either need to buy an arm64 board to test on or emulate. Emulation is slow, and gives a poor idea of performance, however we can test the system for correctness on an emulator before tuning performance on a real system. Since testing on ARM64 requires purchasing a development board, we would need to look at available options. A cursory glance on google indicated a limited availability of options.

6. ARM64 offers more control over the MMU (caching, page sizes, etc).

7. I like arm a lot so far. :)

8. I believe that it’s cheaper to map all of physical ram into the virtual address space on ARM. Possibly due to better PC-relative offsets? Linux does a weird “-2GB” trick to get around this on x86... really fuzzy on the details of this. Anyway, mapping all of physical ram makes memory management a lot easier in a kernel...