Composite Developers Manual

Version 2020-07-04

Gabriel Parmer

Contents

1 Introduction to Composite

This document provides a high-level overview of the Composite OS, and its software infrastructure. Reading this should be sufficient to get a decent understanding how the software is hooked together, and how to productively develop an OS in Composite. It is *not* sufficient to change any of the fundamental structure of the system, but quite a bit can be accomplished by staying within the default parameters of the system.

1.1 Why a New OS?

Monolithic OSes have been amazing catalysts for the software progress we've seen since punch cards. Most OSes are actually moving *away* from the traditional monolithic structure. Linux continues to increase the number of drivers and services exported to user-level, and Android places a focus (through `binder`-based IPC) on user-level services; OSX still supports Mach APIs (a first generation microkernel) along-side its BSD personality; and Windows has been leaning into its NT design around user-level personality definition, for example, by integrating VMs into the system APIs (e.g., around the Linux subsystem). `DPDK` and comparable library/driver linkages move I/O to user-level by *bypassing* the kernel wholesale.

Why is this? What is happening here? Different systems make those choices for different reasons, but there are number of consistent trends:

1. Moving functionality that doesn't need to be into the kernel into user-level increases isolation in the system. Less trustworthy services can be placed at user-level, and their failure will not impact the entire system.
2. Performance, especially when using kernel bypass, can be significantly higher when avoiding the general-purpose paths through the kernel. If the kernel has too much overhead, or if the overheads of system calls are significant, directly accessing I/O from user-level is the answer.
3. User-level definition of software enables the decoupling of languages and development environments. Higher-level functionalities (event OS services) can be implemented in higher-level languages, and using the full breadth of useful libraries provided by a modern development environment.
4. Modifying the kernel is hard. The kernel is a relatively adversarial development environment. Software complexity, concurrency, and parallelism are inescapable in this domain, and enabling user-level development of system services side-steps the legacy of kernel complexity.

It is also important to understand how the world has changed. It is unlikely that desktop OSes are going to be replaced at any point. Monolithic systems are simply a good fit for the large functional requirements of users. However, in other domains, the requirements have been changing:

- *Servers: performance.* On the server side, network throughput is approaching that of DRAM, and latency has been precipitously falling, especially with direct RDMA access. The cost of network operations has made kernel overheads (even around the lowly system call) too much of a performance liability.
- *Servers: virtualization.* On the other hand, multi-tenant systems (e.g., computation for rent) require trusted virtualization facilities. Such systems do *not* require a complete monolithic OS to multiplex VMs, and in some ways the software complexity of a full OS is a liability.
- *Embedded and IoT systems.* On the other side of the spectrum, *small* systems that control the physical world around us are becoming more popular, powerful, and feature-rich. Traditionally these systems used a small Real-Time Operating System (RTOS) that focused on simplicity and reliability. Ironically, they were often more dependable while providing *zero* isolation between different tasks (they all ran co-located with the kernel). This makes sense for two reasons:

  1. as the total lines of code in the system were so small that a small development team could reasonably deploy a tested system, and
  2. the interface between software and the surrounding environment was narrowly defined, often around the sensor and actuator data that would traverse a few I^2C lines. Without a user or a network adding complex interactions, the testing space was restricted.

  This has all changed, and the change is most evident in Autonomous Vehicles (AVs). Sensor processing is immensely complex (often using neural networks over many different video and radar feeds), actuators have multiplied and increased in complexity, and the planning of the system must consider many factors, and arrive at a decision within a bounded amount of time. Cars are now some of the most complex software ecosystems that exist – akin to mobile data-centers.

  OSes must be more reliable, fundamentally predictable, and accommodate the consolidation of software that used to be spread across many different computational units, onto shared multi-core hardware. Embedded systems are often multi-tenant systems with much more stringent requirements around timing and reliability.

Aren't microkernels failed technology? For quite a long time, the failure of microkernels has been a meme. However, over the past 15 years, they have proven quite successful. QNX is the underlying OS behind many reliable automotive systems, seL4 variants have driven the secure processor in many iPhone variants, and Minix manages all modern x86 chips (as "ring -3" firmware). As microkernels focus on reliability and minimality, they are popular in security- and dependability-intensive domains that don't see much direct user interaction. The seismic changes in the requirements for systems are only going to increase the trend toward microkernel-like systems.

1.1.1 Why Composite?

Composite is a microkernel in the broadest sense. It breaks OS software into separate page table protected protection domains, and uses IPC for coordination between them. Device drivers and most traditional kernel functionality – including networking, file systems, process management, etc… – execute at user-level.

Composite focuses on the aspects of modern systems that are of increasing importance. These include a pervasive design optimizing for the non-functional properties of the system, and the specialized construction of software systems (from the OS up) by component *composition*.

Non-functional OS requirements. OSes have been designed primarily around providing a set of functionality (e.g., POSIX), and managing resources appropriately. In contrast, Composite is designed to optimize non-functional properties including:

- *Predictable scalability.* As the number of cores increases to N, the cost of system operations should not increase. To *predictably* maintain a given cost independent of N. The goal: If an application can scale to effectively use N cores, it should be able to! This intuitive goal is not provided by existing systems which can be prevented by scalability bottlenecks in the system. A few takes on how Composite provides this can be found in Gadepalli et al. '20, Wang et al. '14, Wang et al. '15, and Wang et al. '16.
- *Resilience to failures.* Software failures are a reality, and must be planned for, and accommodated. In distributed systems, they are explicitly integrated into the design by leveraging redundancy. Composite aims to be resilient to failures by using pervasive isolation boundaries to isolate bodies of software, and system-level techniques to prevent fault propagation. We pioneered some techniques to micro-reboot portions of the system in response to failures very quickly (on the order of 50 microseconds). A few takes on how Composite provides this can be found in Gadepalli et al. '19, Gadepalli et al. '17, Pan et al. '18, Song et al. '13, Song et al. '15, and Song et al. '16.
- *Security via the Principle of Least Privilege (POLP).* The Composite kernel is a *capability-based system* which means that all kernel resources are referenced by per-component tokens. This is the only way to address such resources, thus avoiding the ambient authority inherent in resource access through unrestricted namespaces (e.g., the file system). Components are *specialized* to a task, and their dependencies on other Components in the system are explicit, and controlled. Thus the impact of a compromise is heavily restricted to only those resources of the very specialized compromised component.
- *Predictable end-to-end execution.* Unfortunately, with the increased isolation that comes from raising hardware barriers between different system services and applications, the predictability of execution across multiple components is threatened. How many threads need to be scheduled across all components to enable the computation of a client's reply? Are there reasonable bounds

> on that latency? Composite takes a drastically different approach to scheduling and IPC based on thread migration and user-level scheduling to enable end-to-end predictable execution – tight bounds can be placed on client service requests.

For a more extensive discussion of this, see the position paper.

OS design by composition. Unikernels enable the specialization of system code to a *specific* application, often avoiding system call overheads by going so far as to link the specialized kernel directly into the application. When you specialize for a single application, this can make sense. Composite is motivated by the same factors. A specialized body of system software

- has a lower memory footprint,
- includes only necessary code thus reducing the system attack surface, and
- can be optimized by the compiler for the application's usage patterns (e.g., via partial evaluation and dead-code elimination).

However, Composite aims to enable the composition of a system for multiple applications. In doing so, even the *isolation properties* of the system can be customized, providing strong VM-like isolation where required, and monolithic system-like service-sharing where extensive sharing is required. Much of the focus of the Composite code-base is to decouple various aspects of system software into multiple components so that the composition of these components has maximum customizability.

1.2 Interesting or Novel Aspects of Composite

This subsection will assume some knowledge about existing microkernels, and will conceptually place Composite into this ecosystem.

1.2.1 Thread Migration-based IPC

Inter-Process Communication (IPC) is usually a focal point for microkernels. Most modern (third generation) microkernels use *synchronous rendezvous between threads* to perform IPC. With this, two threads, a *client* and a *server* synchronize to directly pass a limited number of arguments. A sending client waits for a server to receive. Once the server recieves, the client blocks on a receive. The server computes a reply, and sends back to the client, which finally unblocks the client, sending it the reply. In this way, it functionally mimics function call, despite being communication between isolated protection domains. Using efficient primitives (`call` and `reply_and_wait`), this integration only requires two system calls, and two page table switches, which are the dominant costs in modern IPC systems. I'll refer to this IPC mechanism as *traditional IPC*.

Composite instead uses *thread-migration-based invocations*[1] between components for IPC. The classical paper for this is Ford et al., though the *mechanisms* it uses are quite different from Composite's. With thread-migration, the *same thread* executing in a client component *continues* execution in the server in which it invokes a function. To ensure isolation between client and server, the *execution context* (e.g., registers, execution stack, and memory contents) are separated across components, but the *scheduling context* traverses components. This effectively means that scheduling decisions are not required upon IPC, and that server component execution is properly accounted to the clients (even transitively), and scheduled as the clients. For simplicity, I'll refer to this as *invocation IPC* or *Composite's IPC*. The end-to-end (across many component invocations) proper accounting and scheduling of execution is *required* to enable predictable execution.

Ironically, most popular traditional IPC systems (modern L4 variants) have been moving toward thread migration-based IPC. The complexities and sub-optimalities of doing so are summarized in Parmer et al. '10 which is old, but still just as valid.

Composite does not "pay" for using thread migration-based IPC. Composite has the fastest round-trip IPC between protection domains that we know of. We compared against seL4 in Gadepalli et al. '19, which is known to have exceedingly fast IPC.

1.2.2 User-level Scheduling

> TODO

The story of user-level scheduling in Composite is told in four parts:

- The details of how system level scheduling responsibilities over all threads in the system can be exported to user-level, isolated components that can be customized Parmer et al. '08,
- Parmer et al. '10 provides details about how schedulers can be hierarchically composed to delegate scheduling duties closer to applications,
- Gadepalli et al. '17 discusses how many *untrusting schedulers* can predictably coordinate while providing global timing guarantees using temporal capabilities to provide *access control for time*, and
- Gadepalli et al. '20 demonstrates how user-level scheduling can avoid kernel interactions to make Composite scheduling both capable of system-level scheduling, *and* faster than traditional kernel-resident schedulers.

1.2.3 Wait-Free, Parallel Kernel

[1] Note that this is *not* related to the *cross-core thread migration* used by schedulers to move threads between cores over time.

> TODO

The details of the Composite Kernel can be found in Wang et al. '15.

1.2.4 Access Control for Time

> TODO

The details of Temporal Capabilities can be found in Gadepalli et al. '17.

1.2.5 Minimal, Specialized OSes

> TODO

1.3 Composite Development Philosophy

Composite explicitly separates software into four different types:

- The kernel (in `src/kernel/` and `src/platform/`) includes the ~7 KLoC[2] for the kernel. The rest of the Composite code executes at user-level.
- Component implementations (in `src/components/implementation/*/*/`) which include the main software for both system services, and applications.
- Interfaces (in `src/components/interface/*/`) which includes the prototypes for the functional interfaces that link together components, and the stub code which serializes and deserializes arguments for those invocations. Interfaces *decouple* the implementation from the specification, thus enabling the polymorphism required to enable the versatile composition of multiple components together (e.g., a client that depends on an interface, with a server that implements it in a specific way). These can include multiple *variants* which are different means of implementing the interface. This enables library-based implementations for an interface along-side separate service component-based implementations. The appropriate variant can be chosen based on the context of the calling (client) component.
- Libraries (in `src/components/lib/*/`) which can be Composite-specific libraries, and adapted external libraries.

1.3.1 Kernel API

The Composite kernel provides a capability-based API. The high-level ideas:

[2]KLoC = Thousand Lines of Code.

- A component is a combination of a page table and a capability table. A component (through its capability table) can have a capability to a capability table or page table (which we collectively call *resource tables*). That gives it the ability to modify (add, remove, change) the resource table. Thus, components that have these capabilities to resource tables of other components are entrusted to manage them.
- Threads are the abstraction for execution in the system, and begin execution in a component. A capability to a thread denotes the ability to switch to it. Schedulers are the components that require thread capabilities. Where a thread executes (in which component(s)) is unrelated to which component has thread capabilities (scheduler).
- Capabilities to synchronous invocations denote the ability of a component to *invoke* another. This is the basis for dependencies and enabling one component to harness the functionality of another. Importantly, these do *not* transfer much data (4 registers one way, and 3 back), and data transfer must be separately implemented (via interface logic).
- Capabilities to asynchronous activation end-points denote the ability to active a specific thread. However, they are meant to be used for event notification across trust boundaries. TCaps (temporal capabilities) are associated with threads in this case, and they include priority information that can be used to determine if the thread activation should cause a preemption of the current thread. Either way, a notification is added to the thread's scheduler that the activation has occurred. Unless implementing a device driver (which receives asynchronous activations from interrupt sources), or a scheduler, you can likely ignore this kernel resource.

1.3.2 Dependencies

The *dependencies* for each of the bodies of user-level code are *explicit*. Components, interfaces, and libraries each spell out the other libraries and interfaces they require for their functionality. Composition scripts and the `composer` (in `src/composer/`) specify all of the software that should appear in a system image, all of the inter-component dependencies, and which interface variants should be used for each.

When implementing a component, it is important to ask which other libraries and interfaces you want to depend on, and ensure that those dependencies are explicit. If you're implementing a system service, which interfaces do you want to export? Make sure that your component's `Makefile` includes the proper values for this.

1.3.3 Abstraction Hierarchy

The Composite kernel does *not* provide simple interfaces, nor high-level abstractions such as processes, file systems, memory management, and networking. Those functionalities must be built up from components. We're going through our third version of the user-level environment, thus not many high-level functionalities are currently provided. I'll use the system bootup as an example of composing components to increase functionality.

- At system boot-time, only a *single* component is loaded and executed. Your component can be booted this way, in which case it can use the kernel API, and has capability access to all system resources! Generally, this is not all that useful. A good example of a component written in this style is `tests.kernel_tests` which can be executed through the `kernel_test.toml` composition script.
- If you want isolation between bodies of software, the kernel-loaded component can be the loader for other system components. We call a component that loads others a *constructor*. In this case, that component must be trusted to

  1. load in the memory for the components it is constructing,
  2. create a thread in each component (per core), and initialize the components correctly, and
  3. create synchronous invocation resources between components so that they can interact.

  Generally constructors are trusted to perform all operations to ensure the *control flow integrity* of its constructed components. In other words, the initialization entry point, and the synchronous invocation entry points are set up by the constructor, and no other component can alter thread entry points or flow of execution through the component. The goal is that the main logic of constructors should be less than 500 LoC, and, with libraries, < 4 KLoC.

  Components constructed this way, are executed FIFO, in accordance with the initialization procedure (see the chapter on the component execution model). This is generally not all that useful. An example of a system constructed in this style can be found in the composition script `ping_pong.toml`.
- When components wish to more dynamically access system resources (e.g., memory), we can add in a Capability and Memory Manager (which I'll shorthand as *capmgr*). These implement the `capmgr` and `memmgr` interfaces. The capmgr is responsible for receiving requests for system resources (new threads, memory allocations), and tracking those kernel resources. The capmgr is now in chart of initialization order of all components it is responsible for. Unfortunately, this means that those components are still executed FIFO! Also not all that useful: we can do dynamic resource allocation, but execution is still rather constrained. A capability manager can be found in `capmgr.simple`. The `capmgr_ping_pong.toml` composition script is an example of this.
- Finally, if we want configurable scheduling policies, we can add a scheduler into the system (that implements the `sched` interface). The constructor will initialize the capmgr, which will initialize the scheduler, which will then initialize *and schedule* all other components. Schedulers will often use timers to premptively schedule threads in accordance with some policy. An example sched-

uler is `sched.root_fprr` which can be run with the composition script `sched_ping_pong.toml`.

This is a simple example of how layers of abstraction can be added bit-by-bit until some functional and non-functional requirements are achieved. Additionally, new interfaces that implement the specified interfaces could be implemented to provide the functionality in a specialized manner.

1.4 Limitations and Bugs

We want to up-front with the current common limitations of the system. First, you should know that if you see an error involving them, the general techniques to resolve them. Second, you should know about them so that you don't make a design that assumes functionality where there is none. Most of these are *not* systemic issues, and can be solved with enough engineering hours.

In no particular order:

- Composite has a challenging relationship with Thread Local Storage (TLS). Systems typically use a register to hold a pointer to a thread-specific pointer to their storage. On x86-32 this is the `%gs` segmentation register. Due to thread-migration, a single thread executes across components via invocations. We want to avoid having to lookup a per-thread, per-component value on each invocation, so services that are export interface should *not* rely on TLS. Applications should be able to use TLS, but currently there is no organized support. When a new thread is created, `cos_thd_mod(struct cos_compinfo *ci, thdcap_t tc, void *tlsaddr)` should be used to set the TLS register.
- By default, most Linux system calls (accessed through `musl` libc) will *not* work, and will report an error (look at the Linux references to translate the system call number into the functional system call). There is no default POSIX support in Composite, and POSIX support (outside of VMs) will never be a priority. However, restricted support for libc system calls can be added with the `posix` library. That library enables a component to define a set of library-defined system call implementations. This can be used, for example, to provide a tarball-defined RAM file system. Note that even pthreads system calls are not implemented in Composite. See the `sched` interface if you'd like to allocate, modify, and synchronize with threads, or the `sl` library if you are implementing a scheduler (as a library).
- Composite does not support general elf objects. To simplify the kernel, we assume that the program headers of the system component includes only two segments: one that is RO, and one that is RW. The Composite build system (notably `src/components/implementation/comp.ld`) will provide this invariant, but if you try and deviate from that linker, your mileage will vary.

2 Creating and Executing your First OS using Composite

This chapter will introduce you to how to build and run your first OS in Composite. As Composite is based on the component-based construction of OSes, we'll focus on how to "put the legos together", and compose a specialized system from a specific set of components.

It is necessary to understanding a number of concepts:

- **Components.** Components are a combination of a page table and a capability table. They are the unit of isolation in the system and contain and constrain all memory and resource access. Threads can be created in a component and will begin execution at a specified address (like `__start` in SysV). Synchronous invocations to a component also specify entry points for thread-migration-based invocations.
- **Dependencies.** Compilation dependencies are specified for components, interfaces, and libraries. Dependencies are on libraries and interfaces, and the build system will generate the transitive closure of the dependencies.
- **Interfaces.** Similar to interfaces in Java, Composite interfaces decouple functional signatures from the implementation, thus enabling polymorphism of the underlying implementation. They provide all of the logic to serialize and deserialize functional arguments, thus their implementations might rely on other libraries or interfaces. Interfaces can have multiple *variants* that might link a client to their functionality in different ways. This could mean different means of serializing arguments, or implementing some interface as a library!
- **Component invocation.** Synchronous invocation resources in a component's capability table enable it to invoke a specific functional entry point into another component via a thread migration-based invocation. This is the building block of coupling OS service to their clients, and providing system abstraction.
- **Constructors.** Components that are responsible for building other components. They construct the page and capability tables for components, set up the synchronous invocations between them, and initialize any components that depend on them for execution.
- **Capability Managers.** These components are in charge of managing the capability table (and often page-tables) of other components. As such, resources can be allocated, and shared (e.g., shared memory), and later revoked. These components replace the complex capability management in the kernel in most other microkernels.
- **Schedulers.** Perhaps the least surprising: these schedule threads. They manage timer interrupts (thus preemptions), create threads, and implement policies for switching between them to optimize for some goals. The default scheduler is fixed-priority round-robin.
- **Composer and composition scripts.** To create an executable system, specific components, related in specific ways must be "linked" together into a collection of components that each manage some set of resources and provide some abstraction. The *composer* takes a composition

script specification of a given collection of components, and variants for the interfaces that they depend on, and export. It will generate a set of initialization arguments that enable constructors, capability managers, and schedulers to understand how they should manage other components (e.g., how to construct synchronous invocations).

2.1 Creating an executable system

We'll use the `cos` script to simplify using Composite. Our first OS is going to have a constructor, a capability manager, a scheduler, and a simple scheduler test. Specifically, we'll use the following system (from `composition_scripts/sched.toml`):

```toml
[system]
description = "Simplest system with both capability manager and
    scheduler, from unit_schedcomp.sh"

[[components]]
name = "booter"
img  = "no_interface.llbooter"
implements = [{interface = "init"}, {interface = "addr"}]
deps = [{srv = "kernel", interface = "init", variant = "kernel"}]
constructor = "kernel"

[[components]]
name = "capmgr"
img  = "capmgr.simple"
deps = [{srv = "booter", interface = "init"}, {srv = "booter",
    interface = "addr"}]
implements = [{interface = "capmgr"}, {interface = "init"}, {interface
    = "memmgr"}, {interface = "capmgr_create"}]
constructor = "booter"

[[components]]
name = "sched"
img  = "sched.root_fprr"
deps = [{srv = "capmgr", interface = "init"}, {srv = "capmgr",
    interface = "capmgr"}, {srv = "capmgr", interface = "memmgr"}]
implements = [{interface = "sched"}, {interface = "init"}]
constructor = "booter"

[[components]]
name = "schedtest"
img  = "tests.unit_schedcomp"
deps = [{srv = "sched", interface = "init"}, {srv = "sched", interface
    = "sched"}, {srv = "capmgr", interface = "capmgr_create"}]
constructor = "booter"
```

We are loading the `booter` which is a constructor, the `capmgr` which is the capability manager, the sched which is a fixed priority, round-robin scheduler, and a simple schedtest which is running a minimal preemption test. The deps, **implements**, and constructor specifications encode the dependencies for all components, thus the structure of the system.

Lets go through the processes from downloading Composite, to booting and running the system:

```
$ git clone git@github.com:gwsystems/composite.git
$ git branch -f loader origin/loader
$ git checkout loader
```

We should have the loader branch at this point.

```
$ ./cos
Usage:  ./cos  init|build|reset|compose <script> <output name>|run <
    binary>
```

The cos shell script is going to walk us through the execution of the system. The first print out for each cos command is the raw command it is going to execute. You can use these commands to dive into the code appropriately.

First, lets initialize and build the system:

```
$ ./cos init
[cos executing] make -C src config init
...
$ ./cos build
[cos executing] make -C src all
...
```

This will compile the system using all of the default parameters. You'll see some warnings for the c++ standard library, unfortunately. Next, lets compose the specific OS we want to create:

```
$ ./cos compose composition_scripts/sched.toml test
[cos executing] src/composer/target/debug/compose composition_scripts/
    sched.toml test
...
System object generated:
        /home/gparmer/data/research/composite/system_binaries/cos_build
            -test/cos.img
```

This will run our linker that consumes the specific configuration script, and (as specified) generates a system image that packs together a kernel, constructor, and all of the components within that constructor. Now that we have the system image, we can run it!

```
$ sudo ./cos run stem_binaries/cos_build-test/cos.img
[cos executing] tools/run.sh system_binaries/cos_build-test/cos.img
...
(0,8,4) DBG:Test successful! Highest was scheduled only!
(0,8,4) DBG:Test successful! We swapped back and forth!
```

Note that we use `sudo` to run with KVM hardware virtualization support. This makes all of the performance measurements close (but not identical) to what you'd see on the bare-metal. If you remove the `sudo`, you'll use QEMU, software-driven virtualization in which performance is not accurate. We are in the process of standardizing our unit testing frameworks, so please excuse our ad-hoc output.

2.2 Understanding the system binaries

Now lets dive into our executables a little bit. This is useful to understand what's happening in the system, and to be able to debug different parts of the system. All files are created in the `system_binaries/cos_build-test/` directory that was created with that name due to the `test` argument to `./cos run`.

```
$ ls system_binaries/cos_build-test/
constructor  cos.img  global.booter  global.capmgr  global.sched
    global.schedtest  kernel_compilation.log
```

The `cos.img` image is the kernel that has been linked with the constructor binary. The kernel boots up, and creates the `constructor` binary as the first component. The `kernel_compilation.log` file contains the kernel compilation output. Look here if you see errors when you `compose`.

Next, you see a directory for each component in the system named after the `name =` variable names in the composition script. They are named `global.name` as they are all in a "global" scope in the composition script. Note that we don't currently support any other scopes. Each of those directories includes compilation by-products:

```
$ ls system_binaries/cos_build-test/global.sched/
compilation.log  initargs.c  initargs.o  sched.root_fprr.global.sched
```

The binary is the executable for the scheduler. You can introspect on that object using `objdump` and `nm`. For example, see the Debugging section of this document to see how to map a faulty instruction back to the corresponding C. This uses `objdump -Srhtl`. The arguments passed into the component by the `composer` are in `initargs.c`. We won't go over that format here. Last, the `compilation.log` contains the compilation commands and output to aid in debugging.

3 Build System and Software Construction

The Composite user-level has three main bodies of code:

- Components that implement and depend on different functional interfaces.
- Interfaces that provide a functional specification for a given set of functionality, including the serialization and deserialization logic, and potentially different implementation variants.

- Libraries which are statically linked into components.

Their structure departs from a POSIX organization, and is more similar to language-specific dependency management. However, because this dependency management is at the level of the C ABI, it should be portable across any languages that FFI to C (i.e. most of them).

3.1 Problems Solved

The problems solved by the library system in Composite include:

- We want to be able to share library functionality between different components.
- As such, components must somehow understand which directories to use in their include paths, library paths, and with which libraries to compile.

Composite does *not* solve the following issues:

- We do not attempt to solve library versioning through dependency management. Instead, we maintain a versioned mono-repo for all standard libraries, and expect external libraries to be bound to a version. This does *not* scale, but is reasonable at our current development team size.
- Formulating a coherent and complete specification for dependencies. We integrate dependency specification into Makefiles, and enable adapters between the Composite build system, and external libraries.
- Dependency minimization is not provided. If a component, interface, or library specifies a set of dependencies, Composite provides no means to ensure that set is minimal. As we use static libraries (`lib*.a` files), only `*.o` files that satisfy undefined symbols are included.
- Libc is special-cased in the system. We use `musl` libc, and ensure that it is compiled into components **after** all other libraries and interfaces are linked to. This is required to ensure that all necessary libc functionality is pulled in (i.e., that the linking processes sees the undefined symbols from other libraries and interfaces).

3.2 Compilation Overview

There are six main `make` rules to guide system construction.

- `make config` - This is run once, and first. It selects the architecture.
- `make init` - This builds

    1. libraries that do *not* want to pay the cost of potential recompilation during the normal development cycle, compile here.

2. libraries that have an initialization procedure that generate include files must be run before other libraries that depend on them.

- `make` or `make all` - The build command for the regular development cycle. Builds all libraries, interfaces, and then components, based on normal `make` build dependencies.
- `make clean` - Cleans and removes the compilation by-products of much of the implementation, aside from the state that was built in `init`.
- `make distclean` - Cleans all compilation by-products including those constructed by `init`.
- `make component` This is the main mechanism to create components that are statically linked with a specific set of libraries, and interface *variants*, and with a given set of initial arguments that are used to create a set of components in a specific system image. This rule is used by the automatic compilation process of the `composer` to construct a customized system image.

The Composite compilation process assumes that

- We do *not* use system includes (as we provide all relevant code), and do not use dynamic linking. Thus there should *not* be a used Procedure-Linkage-Table (PLT) in the executables.
- Position-Independent Code (PIC) is not used as we're currently statically linking all executables. Because of this, the Global Offset Table (GOT) should not be used in executables.
- Similarly, Position-Independent Executables (PIE) are not used.

References:

- PIC introduction.
- Some internals.

4 Dependency Specifications for Compilation

Dependencies are specified for components, libraries, and interfaces. To avoid ambiguity, we'll describe a component, library, or interface to be a software *blob*. A blob's dependencies are the set of libraries and interfaces required for its proper execution. Interfaces and libraries also *export* functionality for use by other software blobs. Linking these software dependencies (between a dependent software blob, and the interface or library that exports the functionality) requires that the build system understand the include paths, library paths, and library outputs.

This document discusses the means of concretely resolving these dependencies, and focuses on the *build system*'s dependency resolution integration. The dependencies between *components* for a specific system graph (i.e. the structure of components linked together via interfaces) is covered separately when discussing *composing* a system.

4.1 Dependency Specifications

Dependencies are specified in the top-level `Makefile` for the specific component (or library or interface). These are in `src/components/lib/*/Makefile` for libraries, and `src/components/interface/*/Makefile` and `src/components/interface/*/*/Makefile` for interfaces[3]. The required `Makefile` variables are discussed in the following sections.

These specifications culminate in the following dependency relations within the core Composite software.

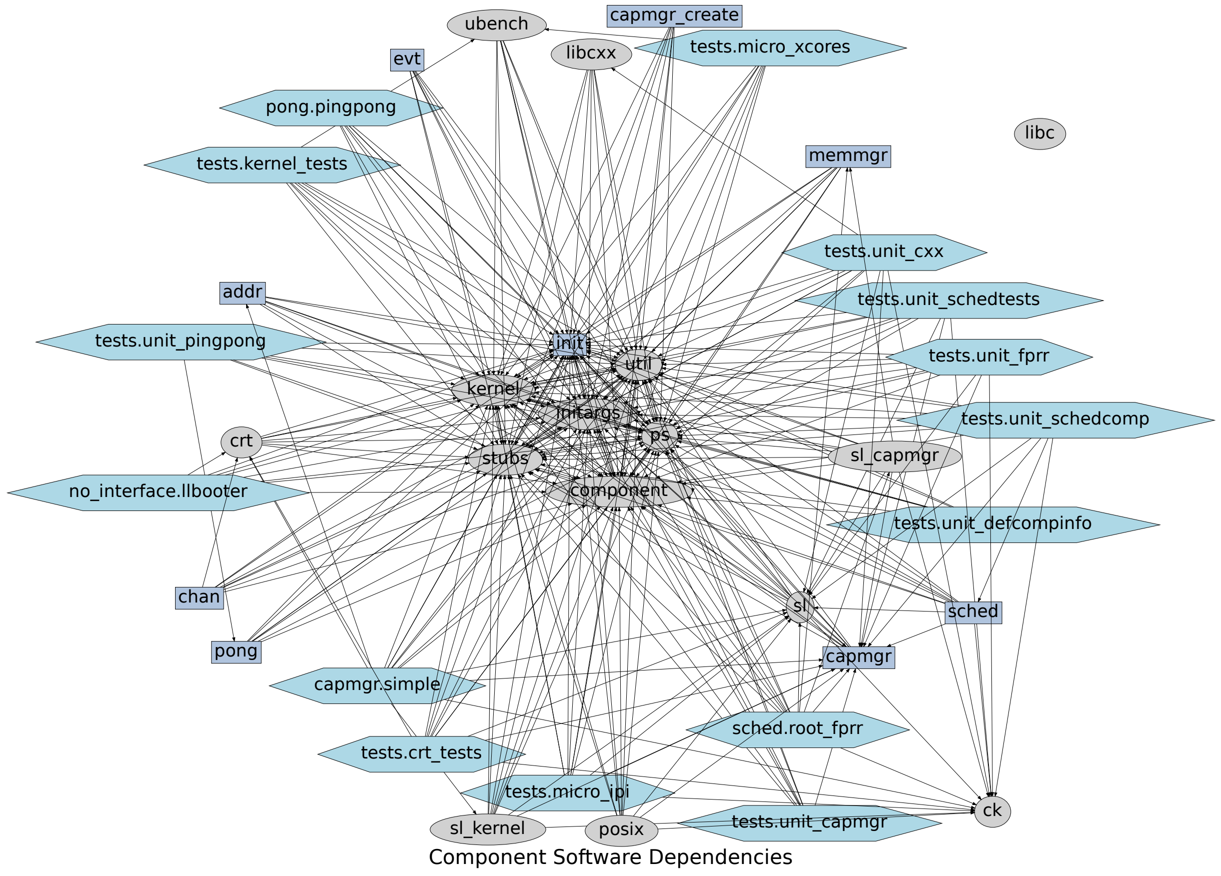

Figure 1: Software dependencies in Composite. Teal hexagons are *component* implementations, slate rectangles are *interfaces*, and gray ellipses are *libraries*.

[3]Note that currently, the second path is largely ignored.

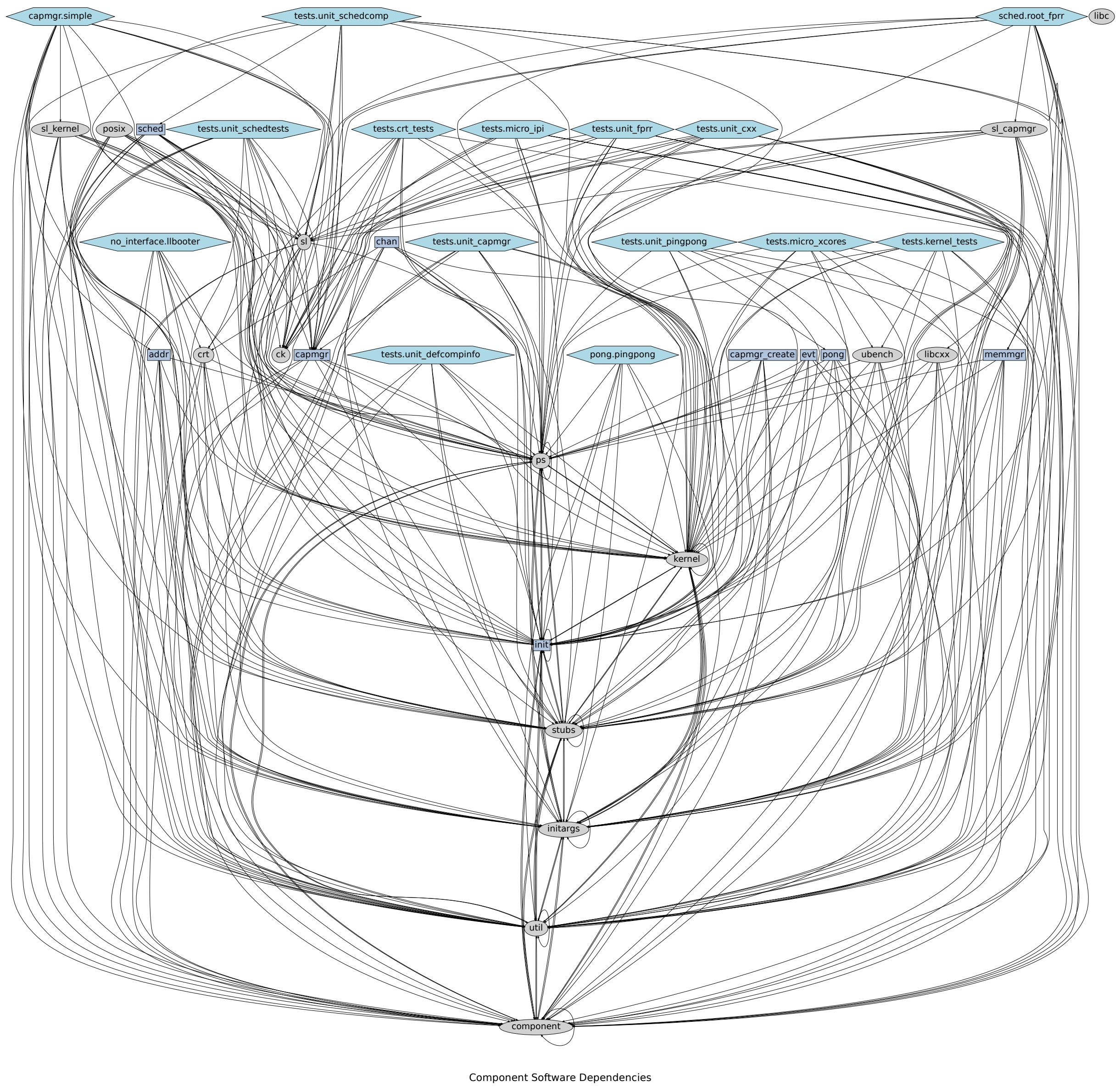

Figure 2: A different rendering of the software dependencies in Composite. Teal hexagons are *component* implementations, slate rectangles are *interfaces*, and gray ellipses are *libraries*.

4.1.1 Interface and library dependencies

The core dependencies of a blob include the interfaces and libraries for which it requires `.h` and object files to compile. These are all specified using the interface and library names, not the names of specific files generated by the libraries or interfaces.

The list of interfaces is a space separated list:

```
INTERFACE_DEPENDENCIES = init
```

The list of libraries is also a space separated list:

```
LIBRARY_DEPENDENCIES = component kernel
```

Specifying these will automatically set up the `-I` paths, the `-L` and `-l` paths corresponding to the depended on libraries and interfaces, and will compile with any mandatory libraries appropriately.

All of the following must specify these dependencies:

- component implementations,
- interfaces, and
- libraries.

Notes:

- Most components depend on the `init` interface to denote which component should provide their initialization. The depended-on component is often the scheduler, but might be the capmgr, or the constructor if the client is very low-level.
- Most components depend on the `capmgr_create` interface, as it denotes which component is the capability manager that is allowed to manage their kernel resources.

4.1.2 Library and interface object outputs

Most libraries, and some interfaces generate static libraries (as `lib*.a` archives of `*.o` files) that are linked in with clients to satisfy undefined symbols. The names of the output library files are specified as:

```
LIBRARY_OUTPUT = (libname|ifname)
```

In such a case, either a `liblibname.a` or a `libifname.a` file is looked for when satisfying dependencies.

Mandatory object libraries. Some libraries and interfaces output a `*.o` object file that is *mandatory linked* with clients. This is *rare* as normally you only want to link with a client if the client has undefined symbols satisfied by you, in which case static libraries are the correct solution. However, some libraries provide boot-up code, and other code that does not satisfy an undefined symbol within the client, instead providing necessary code that either provides data-structures, or invokes the client code. If you aren't sure, you should *not* use mandatory object files. A library's output file is specified as follows:

```
OBJECT_OUTPUT = libfile
```

This will create the file `libfile.lib.o` which will be mandatorily linked into any dependent blob.

These specifications are applicable to

- libraries, and
- interfaces.

Additionally, libraries must specify their list of include paths. Each library should provide a number of `.h` files to specify their functional types, and many libraries might impose their own location for those headers. Thus, libraries can provide a relative specification for where those paths are (in a space-separated list):

```
INCLUDE_PATHS = .
```

This specification is applicable only to

- libraries.

Notes for library output:

- The names all correspond to the logical names of the libraries. For example, `component`, not `libcomponent.a` or `component.lib.o`.
- The build processes is assumed to not generate files with identical names to those generated by the build process.

4.1.3 Component exports

Components are often specifically implemented to provide one or more interfaces. These interfaces are typically specified in a space-separated list:

```
INTERFACE_EXPORTS = init sched
```

This specification will force the component to be linked with the corresponding server stubs for the corresponding interfaces, and will include them in the compilation `-I` paths.

This specification is applicable only to

- components.

4.2 Example specifications

> TODO

4.3 Debugging

Most bugs with mis-specification center around dependencies and exports can be debugged using a few mechanisms.

1. If there is a compilation error around not finding a `.h` file, then there are likely missing dependencies. Figure out where the missing `.h` file is, and add the corresponding dependency.
2. If there are unresolved symbols, viewing the compilation logs, and use `nm` or `objdump` to determine which symbols are unresolved, and from which object file. Then trace back to figure out which exports or dependencies are missing.
3. Look at the specific `-L*`, `-l*`, and `-I*` values in the compilation process, and verify that they line up with your expectations (i.e., they aren't missing specific expected dependencies).

5 Libraries and Interfaces

Libraries are not very different from those in other systems, but as dependencies are explicit in Composite, their structure within the surrounding software is different. Interfaces abstract the functional communication between components that form the core dependencies between components.

5.1 Libraries in Composite

Libraries are initialized during `make init` and built during `make`. Libraries in `src/components/lib/` are built in an undefined order, thus requiring the decoupling of `make init` which builds header files where necessary for specific libraries (for exampele, `ps` and `libc`), and `make` which allows each library to `#include` each other (thus requiring all headers to be present).

The build system guarantees the following:

- For a component, library, or interface, the dependencies for libraries are properly compiled and linked – the corresponding include paths, library paths, and library objects are added into component compilation.
- For a library, its include paths and objects are made visible through the build system.
- All of these rules apply transitively. Libraries and interfaces required by libraries and interfaces, and so on.

5.2 Building a Composite Library

5.3 Creating a New Library

A script helps us make a new library:

```
$ cd src/component/lib/
$ sh mklib.sh name
```

Creates a new library directory in `src/component/lib/name/`. More details on libraries can be found in the corresponding chapter.

5.4 Creating a New Interface

Same story for interfaces:

```
$ cd src/component/interface/
$ sh mkinterface.sh name
```

Which will create a new interface in `src/component/interface/name/`.

5.5 Integrating an External Library

5.5.1 Considerations

If there are automatically generated header files, then those

5.6 Debugging

Thread Local Storage (TLS).

5.7 FAQ

6 Components

6.1 Execution Model

A component goes through a sequence of initialization functions when it starts up. Whereas System V defines `__start` to execute constructors, then `main`, the initialization sequence in Composite must consider two additional constraints:

- The dependencies between components paired with the thread migration mean that there is a required initialization sequence. If two components C and S (client and server) have a dependency from $C \rightarrow S$, S must initialize before the client. Were this not the case, the client, C, could invoke S *before* it is initialized. Thus the initialization sequence must facilitate this synchronization.
- Service components must be able to leverage parallel execution. Whereas monolithic kernels provide APIs to create execution another core – either passively through thread migration, or intentionally through by controlling affinity, the Composite kernel is far lower level, and such abstractions must be *created* via Component functionality. Thus, parallel execution is explicit through initialization, and services can execute on each core available to them.

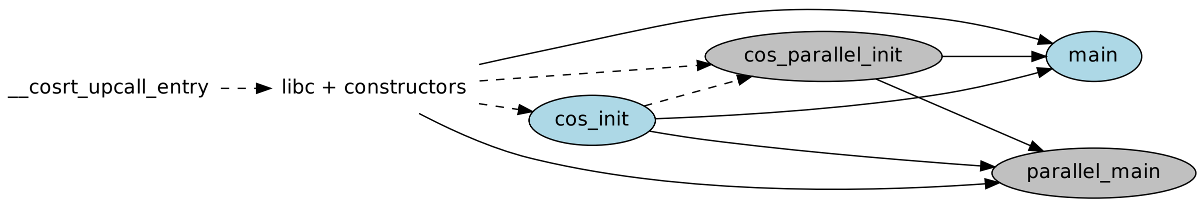

Figure 3: Execution begins at `__cosrt_upcall_entry` which kicks off the sequence of initialization functions for a component. Gray components run in parallel on each core, and blue run sequentially. A given function is only called if it is defined by your component, and the longest path is taken. If both `main` and `parallel_main` are defined, then only `parallel_main` is executed. Dashed transitions prevent client execution; only solid transitions allow clients to execute.

Composite splits initialization into a variety of phases, and a component must *opt-in* to a specific initialization phase by defining the corresponding function. Specifying a dependency on the `component` library is sufficient to enable this functionality. The sequence of these calls is depicted in the Figure above. We detail the functions below.

- `void cos_init(void);` - The first initialization function executed executes on the initialization core. When this returns, the system either continues execution with `cos_parallel_init`, or continues to one of the mains.
- `void cos_parallel_init(coreid_t cid, int init_core, int ncores);` - Each core on which the component executes has a thread that executes through this function. The id of the core `cid`, if the current invocation is the initial core (`init_core`), and the number of cores (`ncores`) are passed separately on each core. Threads barrier synchronize before this function is executed, so `cos_init` has completed execution by the time *any* thread executes this function.

After the previous functions have executed, the system will initialize this component's clients. Thus, it is essential that all necessary initialization to receive invocations from client is completed before returning from the previous functions.

- `int main(void);` - This mainly exists to support legacy execution. The current system does *not* pass arguments, nor environmental variables. This function is called on the same care (in the same thread) as called `cos_init` and `cos_parallel_init(_, 1, _)`.
- `void parallel_main(coreid_t cid, int init_core, int ncores); -` Similar to `cos_parallel_init`, this is executed on each core, but done in parallel with client initialization and execution. For services that require persistent execution, this provides it.

6.2 Abstractions - Execution

6.2.1 Kernel API

6.2.2 Scheduling Library

6.2.3 Scheduler Component

6.3 Abstractions - Kernel Resources

6.3.1 Direct Capability Management

6.3.2 Capability Manager

6.4 Abstractions - Communication

6.4.1 Synchronous Invocations

6.4.2 Asynchronous Activations

6.4.3 Channels

6.5 RTOS

6.6 Creating a New Component

All components exist in the `src/component/implementation/` directory. To create a new component, we need to first answer which is the dominant interface it exports. We either create, or make a new directory for that *interface*, and within it, create an appropriately-named directory for the component. If it is an application, thus does *not* export an interface, then we use the `no_interface/` directory.

We can copy in the Makefiles and template from the `skel/` directory. Luckily, a script will do all of this for us:

```
$ cd src/component/implementation/
$ sh mkcomponent.sh sched edf
```

Which would create a component in `src/component/implementation/sched/edf/` for us, and hook it into the build system.

7 System Composition

7.1 Composition Scripts

8 Debugging

8.1 Print-based debugging

Though less than idea, print-based debugging is a conventional base-line. Generally, you want to make sure that you use `printc` as it most directly prints out. However, on real hardware, it uses serial, which has a very low throughput. For this reason,

1. printing can significant change the non-functional behavior of the system, and
2. you must make an attempt to minimize the amount of data printed out.

A general technique that is *necessary* for benchmarks, and can be useful for some debugging, is to batch data to print, and actually send it to `printc` *after* the phenomenon you're measuring has completed.

8.2 What code caused a fault?

If your code experiences a fault, how do you figure out what code caused the fault? The fault will be reported as such[4]:

```
General registers-> EAX: 0, EBX: 0, ECX: 0, EDX: 0
Segment registers-> CS: 1b, DS: 23, ES: 23, FS: 23, GS: 33, SS: 23
Index registers-> ESI: 0, EDI: 0, EIP: 0x47800037, ESP: 40810ed8, EBP:
    40810f10
Indicator-> EFLAGS: 3202
(Exception Error Code-> ORIG_AX: 6)
FAULT: Page Fault in thd 2 (not-present write-fault user-mode  ) @ 0x0,
     ip 0x47800037
```

[4]Note that this is a page-fault (access to undefined memory), but faults will print out for other faults as well.

This says that the fault happened while executing in thread number 2. The registers of that thread when the fault occurred are printed as esi through ebp. eip is the instruction pointer.

Let's say this fault was in a cpu component. We wish to figure out which line of code in the *.o component caused the fault. For this we use objdump, a program that allows you to decompile a component, and look at its assembly source. If the C code was compiled with debugging symbols by using the -g compiler flag (which we do by default for Composite), then it will also show C lines interwoven with assembly. To make this code maximally readable, ensure that src/components/Makefile.comp includes:

```
OPT= -g -fvar-tracking
#OPT= -O3
```

If optimizations are used (e.g., -O3), the code is mangled in the name of efficiency. To ensure that all code (outside of libc) uses these flags, make sure to make clean; make.

In the root directory, execute the following:

```
$ objdump -Srhtl c.o
```

(where c.o is the component in the build directory for the component).

You'll see the contents of the object file. We know that the fault happened at instruction address 0x47800037 from the fault report. Here we ignore the top bits of the address (as the component is offset into virtual memory), and do a search through the object file for the instruction addressed 37. We find the code:

```
void cos_init(void *arg)
{
  20:   55                      push   %ebp
  21:   89 e5                   mov    %esp,%ebp
  23:   83 ec 08                sub    $0x8,%esp
/.../spin.c:18
        assert(0);
  26:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
                        29: R_386_32    .rodata
  2d:   e8 20 00 00 00          call   52 <prints>
  32:   b8 00 00 00 00          mov    $0x0,%eax
  37:   c7 00 00 00 00 00       movl   $0x0,(%eax)
/.../spin.c:20
//      spin_var = *(int*)NULL;
        while (1) if (spin_var) other = 1;
  3d:   a1 00 00 00 00          mov    0x0,%eax
```

We can see that instruction 37 dereferenced a NULL pointer. More importantly, if you look up in the code, we see that the code corresponds to line 18 in /.../spin.c which corresponds to

```
assert(0);
```

That line is within the `cos_init` function. Because we compiled our components with debugging symbols, we can see the C code. So we can see that the assert function caused this error. Now you see the usefulness of assertion statements. Instead of going through this whole process with objdump, you could have simply looked up in the log and found the following line:

```
assert error in @ spin.c:18.
```

Much easier than disassembling objects. However, when a fault is caused by an error that didn't trigger an assert, you must use the above techniques to track down the error.

8.2.1 Special faults

Look at the registers and code carefully when you get a fault. Two specific cases, and their cause:

1. If a register includes the value `0xdeadbeef`, you can search through the Composite code to see how this could be caused. The most common cause of this is that you've *returned* from a thread that was created in a component. `sl` does not provide graceful thread teardown by default.
2. If the assembly instruction that is faulting is doing an operation using memory indexed by `%gs`, then you have a Thread Local Storage (TLS) issue. We do not, by default, set up TLS, and you can manually do that when new threads are created using `cos_thd_mod`. If you're using a library that relies on TLS in a service (not an application) component, support might be quite involved. If you can disable TLS through the configuration of the library, that is the first priority. Otherwise, talk to the core Composite team.

9 Testing

> TODO

10 Component Documentation

This section pulls in the documentation from each component. You can find the headers and code for the components in `src/components/implementation/*/*/`.

10.1 capmgr.simple

A capability manager that focuses on simplicity.

10.1.1 Description

The main area of simplification for this component is in statically limiting the number of delegations for each resource. We can allocate a maximum, predefined, number of resources, and share them a maximum number of times. Revocation will remove up to that many shared references. This results in an implementation that is more amenable to analysis, but might be too limiting for larger systems. It is somewhat surprising how far you can get with maximum three delegations, however in an RTOS. The intuition here is that if we are mainly using channels for communication, they are most often one-to-one (SPSC), thus require sharing only between the channel manager, and two clients.

10.1.2 Usage and Assumptions

- Assumes that the capability image upon initialization includes all resources that we'd have as if we were boot up on directly on the kernel. This includes resource table references to ourselves.
- Also assumes that `initargs` have been set up by the `composer` to tell us where the capabilities are that correspond to our clients.
- Assumes a maximum number of resources, and delegations for those resources.

10.1.3 Dependencies and Exports

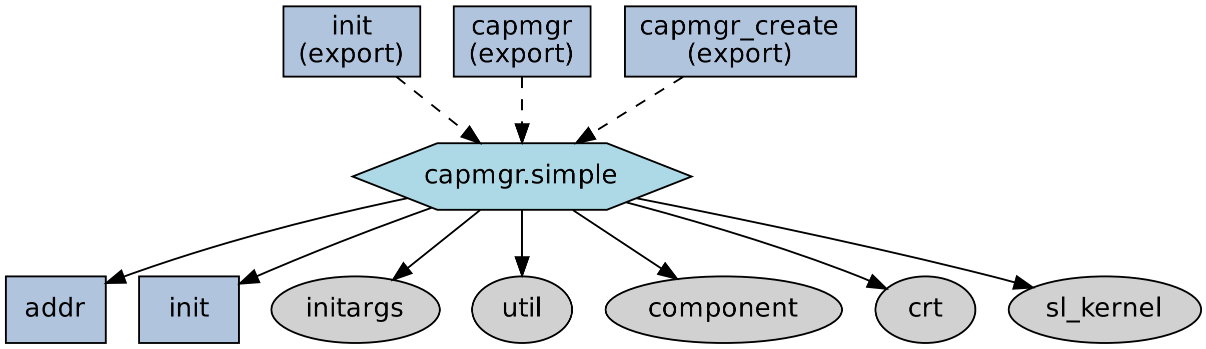

Figure 4: Exports and dependencies for capmgr.simple. Teal hexagons are *component* implementations, slate rectangles are *interfaces*, and gray ellipses are *libraries*. Dotted lines denote an *export* relation, and solid lines denote a *dependency*.

10.2 sched.root_fprr

This is the most common scheduler we use in Composite that implements fixed-priority, round-robin (within the same priority) scheduling.

10.2.1 Description

Pulls most of the scheduling logic in from the `sl` and `sl_capmgr` libraries.

10.2.2 Usage and Assumptions

- We assume that this will execute *on top of* the `capmgr`.
- All components that depend on this component for `init` will be scheduled by it.
- Components dependent on this for scheduling, *must* also depend on the capability manager for `capmgr_create` so that it is allowed to create threads in them.

10.2.3 Dependencies and Exports

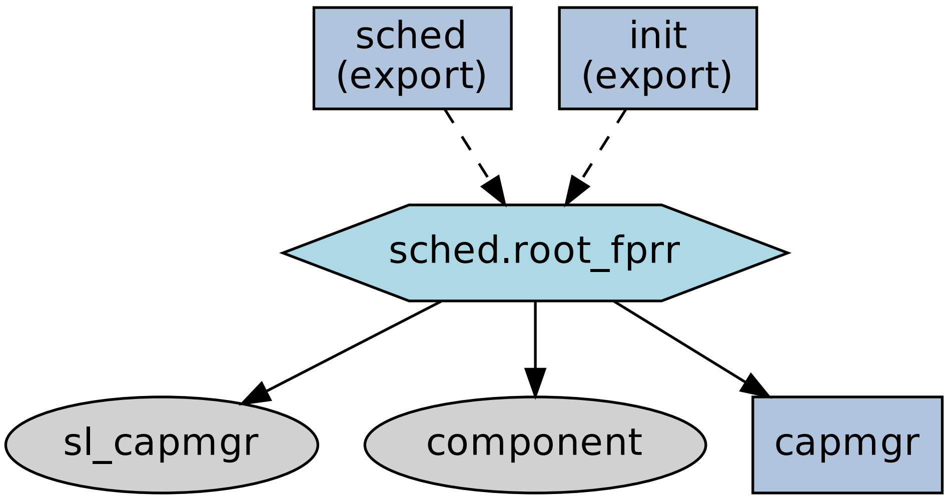

Figure 5: Exports and dependencies for sched.root_fprr. Teal hexagons are *component* implementations, slate rectangles are *interfaces*, and gray ellipses are *libraries*. Dotted lines denote an *export* relation, and solid lines denote a *dependency*.

10.3 no_interface.llbooter

A *constructor* implementation that is as simple as possible, and pushes most complexity to shared (and better tested) libraries such as `kernel` and `crt`.

10.3.1 Description

Creates a set of components as provided by the `composer`. Takes a tarball of these components (accessed through `initargs`), and an `initargs` specification of those components to be booted. This will FIFO schedule initialization through the components that depend on it for `init`.

10.3.2 Usage and Assumptions

- Strongly assumes that the `composer` is used to provide all necessary metadata to construct the rest of the system.
- We currently define the `addr` interface that is a hack to provide frontier (heap pointer) information. This should be replaced with additional `composer` information being passed by `initargs` eventually.

10.3.3 Dependencies and Exports

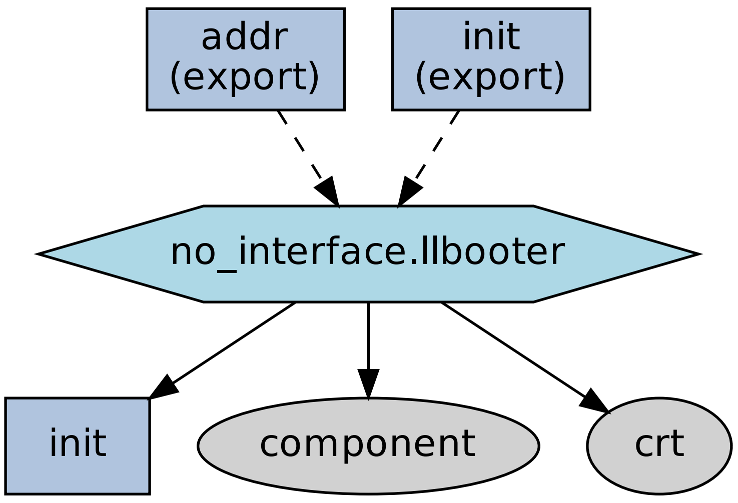

Figure 6: Exports and dependencies for no_interface.llbooter. Teal hexagons are *component* implementations, slate rectangles are *interfaces*, and gray ellipses are *libraries*. Dotted lines denote an *export* relation, and solid lines denote a *dependency*.

11 Interface Documentation

This section pulls in the documentation from each interface. You can find the headers for the interface in `src/components/interface/*/`.

11.1 sched

Scheduler interface that enables thread creation, synchronization, and timing.

11.1.1 Description

Components that implement this provide scheduling services, and implement some scheduling policy. Most schedulers are going to depend on the `sl` and `sl_capmgr` libraries.

11.1.2 Usage and Assumptions

- It is quite common for components to depend on both `sched` and `init` if they want the scheduler to initialize them, and to schedule them.
- Most scheduler implementations will also require you depend on the `sl` libraries, and the current software abstractions depend on a `capmgr` for thread creation.

11.1.3 Dependencies and Exports

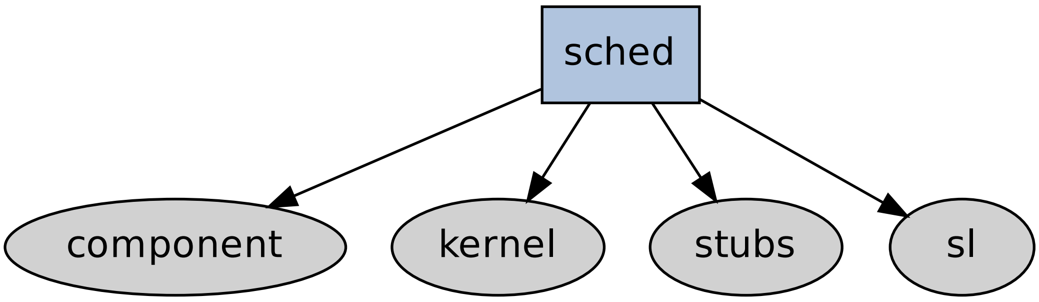

Figure 7: Exports and dependencies for sched. Teal hexagons are *component* implementations, slate rectangles are *interfaces*, and gray ellipses are *libraries*. Dotted lines denote an *export* relation, and solid lines denote a *dependency*.

11.2 capmgr_create

11.2.1 Description

This interface does *not* provide significant functionality. Its function should *not* be invoked, and if it is, it should return with no side-effects.

11.2.2 Usage and Assumptions

This interface is used as a `composer` signal that a client of the interface can be managed by a specific capability manager. This is necessary for the `composer` to create the permissions in the capmgr to be able to create a thread in the client of this interface. `capmgr_create` to a `capmgr` is often paired with a `init` to a `sched` (that depends on the same `capmgr` for `capmgr_create` and `capmgr`).

11.2.3 Dependencies and Exports

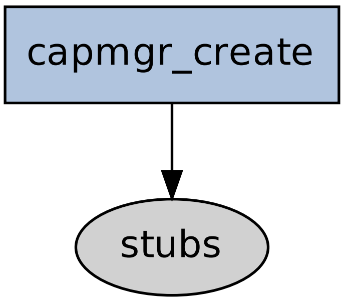

Figure 8: Exports and dependencies for capmgr_create. Teal hexagons are *component* implementations, slate rectangles are *interfaces*, and gray ellipses are *libraries*. Dotted lines denote an *export* relation, and solid lines denote a *dependency*.

11.3 chan

Channels for asynchronous communication both within a component, and between components.

11.3.1 Description

Provide bounded-buffer, asynchronous channels to send generic data from senders to a receiver. It is *asynchronous* in three different ways

- if a send is made to a channel and there are empty slots, the data is added, and the send returns immediately (i.e., it is asynchronous),
- if a receive is made and there is data available to read out of the channel, the data is immediately returned, and
- both send and receive take a parameter that forces asyncrony even if the previous cases are not true.

The channel allocation is *decoupled* from the send and receive endpoints (hence-forth "endpoints") that are the targets for sending and receiving. Having a channel imbues the ability to create endpoints. Each endpoint, and the channel act as a reference that must be destroyed to release channel resources.

11.3.2 Usage and Assumptions

Channels have a number of variants, that are configured when creating the channel. The variants include

- the size of the buffer, and how large each item are in the channel,
- if the channel is single-producer, single-consumer (SPSC), or multi-producer, single-consumer (MPSC), and
- if the channel has exactly the number of items specified when creating the channel, or if it can be optimized by having more.

This is specified in the flags as such:

```c
typedef enum {
    CHAN_DEFAULT   = 0,
    CHAN_MPSC      = 1,      /* !CHAN_MPSC == SPSC */
    CHAN_EXACT_SZ  = 1<<1,   /* The channel size cannot be higher than
       its initialization size */
} chan_flags_t;
```

The default usage is SPSC, while giving the runtime leeway to use larger channel sizes than specified.

The creation API includes the necessary ability to create send and receive end-points that can be used for their corresponding operations. Channels are *only* useful to create send and receive end-points. If the channel is SPSC, then only a single sender and receiver can be created; if it is MPSC, multiple senders can be created.

Deallocation. A channel rcv can only be closed if it has been drained. Once a channel has been destroyed, or the rcv has been closed, then all future sends should return an error. Each `chan_snd`, `chan_rcv`, and `chan` represents a reference, and when the reference count goes to zero, the channel is deallocated.

11.3.3 Dependencies and Exports

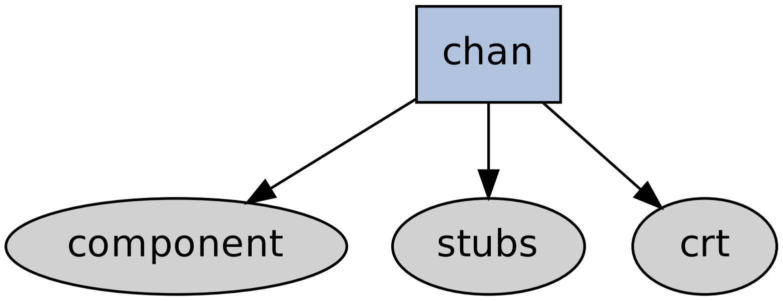

Figure 9: Exports and dependencies for chan. Teal hexagons are *component* implementations, slate rectangles are *interfaces*, and gray ellipses are *libraries*. Dotted lines denote an *export* relation, and solid lines denote a *dependency*.

11.4 init

The interface used for component initialization.

11.4.1 Description

Not only does this functionally enable the synchronization necessary for the dance between `cos_init`, `cos_parallel_init`, and `main` or `parallel_main`, it also is a *marker* used by the `composer` to enable the server providing this interface the resources and `initarg` values to be able to conduct this initialization.

11.4.2 Usage and Assumptions

Any component that wishes to be initialized, *must* depend on this interface. Generally, you should depend on the "most abstract" provider of the interface. In a system that includes a constructor, a capability manager, and a scheduler, you should generally depend on the scheduler for `init`. That way, you'll be scheduled using a traditional policy, and not non-preemptive FIFO (the policy for the constructor and capability manager). Additionally, any capability manager must be enabled to create threads within a client of this interface, so an `init` to a `sched` should be paired with a `capmgr_create` to the `capmgr`.

11.4.3 Dependencies and Exports

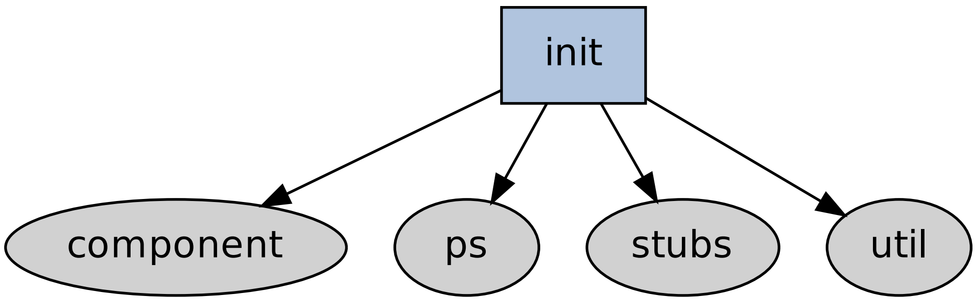

Figure 10: Exports and dependencies for init. Teal hexagons are *component* implementations, slate rectangles are *interfaces*, and gray ellipses are *libraries*. Dotted lines denote an *export* relation, and solid lines denote a *dependency*.

11.5 capmgr

The capability manager API that provides dynamic resource allocation, delegation, and revocation.

11.5.1 Description

A component that is an implementation of this interface is trusted to add and remove resources from the resource tables of components that depend on the `capmgr` interface.

11.5.2 Usage and Assumptions

Any component that implements this, should also implement `capmgr_create` that enables even components that don't want to invoke any of the functions in this interface (as they are relatively static) to enable the `capmgr` to create threads within them (often in response to creation requests from a `sched`).

11.5.3 Dependencies and Exports

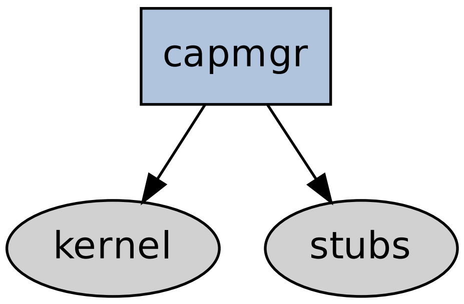

Figure 11: Exports and dependencies for capmgr. Teal hexagons are *component* implementations, slate rectangles are *interfaces*, and gray ellipses are *libraries*. Dotted lines denote an *export* relation, and solid lines denote a *dependency*.

11.6 evt

Event notification API. Traditional event APIs such as `select`, `poll`, `epoll`, and `kqueues` aggregate multiple event sources (sockets, files, pipes, timers, signals, etc...), and enable an application to wait for an event on any one of them. This API provides a generic mechanism for resource managers (e.g., for channels or timers) to *trigger* generic events, and for applications to *wait* for any of those events. The API at its core is very simple:

- create (and destroy) event end-points,
- add generic event sources to them,
- wait for an event on any of them, and
- have resource managers trigger events.

Any complexity in the implementation comes from attempting to optimize in a number of dimensions:

- We want to batch notifications to the maximum degree possible, and enable a *waiting* thread from even invoking the manager if there are pending events.
- For events that can be triggered without involving the corresponding resource manager (e.g., channels), we'd like to avoid invoking the manager to correspondingly trigger the event.

Note that channels are designed to provide both of these optimizations, thus the merging of the `evt` API with channels.

11.6.1 Description

11.6.2 Usage and Assumptions

11.6.3 Dependencies and Exports

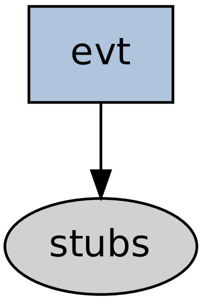

Figure 12: Exports and dependencies for evt. Teal hexagons are *component* implementations, slate rectangles are *interfaces*, and gray ellipses are *libraries*. Dotted lines denote an *export* relation, and solid lines denote a *dependency*.

12 Library Documentation

This section pulls in the documentation from each library. You can find the headers for the libraries in `src/components/lib/*/`.