Customizing a subnet

Tutorial to add a new syscall to Filecoin Virtual Machine and create a new built-in actor in IPC to activate the syscall IPC is uniquely hyper customizable as a scalability framework. Subnets are highly customizable and can be temporal, allowing subnet operators to spin up customized subnets for various needs, including modular consensus, gas option, configurable chain primitives, customized features vis pluggable syscalls and build-in actor.

In this tutorial, we will focus on how to extend features to your IPC subnet by customizing syscalls that are 'pluggable' as needed.

IPC uses theFilecoin Virtual Machine (FVM) as its execution layer, which is a WASM-based polyglot VM. FVM exposes all system features and information through syscalls, as part of its SDK. The FVM SDK is designed to be pluggable to enable user-defined custom features with syscalls, while implementing default features from FVM kernel. Use cases includes:

- Extending chain-specific syscalls once IPC supports more root chains. Because other chains may have their own special syscalls different as Filecoin (proof validation, etc.).
- Extending features to support better development tools. E.g. adding special debugging syscalls, adding randomness syscalls, and supporting more ECC curve, etc.
-

Pre-requisite knowledge for tutorial:

- ref-fvm
- fvm syscall APIs
- IPC/fendermint
- implementation
- Rust
- Ambassador
-

Instructions

These instructions describe the steps required to create a new kernel which implements a new syscall along with an example built-in actor that shows how you would call that syscall. Full examplehere .

TIP: For clarity, the instructions may have skipped certain files (like longCargo.toml files) so make sure to refer to the above full example, if you want to follow along and get this compiling on your machine.

1. Define the custom syscall

2. In this example, we will be creating a simple syscall which accesses the filesystem. Inside syscalls, you can run external processes, link to rust libraries, access network, call other syscalls, etc.

3. We'll call this new syscallmy_custom_syscall
4. and its defined as follows:
5. ```
6. Copy
7. pubtraitCustomKernel:Kernel{
8. fnmy_custom_syscall(&self)->Result;
9. }
10. ```
11. fendermint/vm/interpreter/src/fvm/examples/mycustomkernel.rs#L23
12. Define a structCustomKernelImpl
13. which extendsDefaultKernel
14. . We use theambassador
15. crate to automatically delegate calls which reduces the boilerplate code we need to write. Here we simply delegate all calls to existing syscall to theDefaultKernel
16. .
17. ```
18. Copy

19. # [derive(Delegate)]

20. # [delegate(IpldBlockOps,where="C: CallManager")]

21. **[delegate(ActorOps,where="C: CallManager")]**
22. **[delegate(CryptoOps,where="C: CallManager")]**
23. **[delegate(DebugOps,where="C: CallManager")]**
24. **[delegate(EventOps,where="C: CallManager")]**
25. **[delegate(MessageOps,where="C: CallManager")]**
26. **[delegate(NetworkOps,where="C: CallManager")]**
27. **[delegate(RandomnessOps,where="C: CallManager")]**
28. **[delegate(SelfOps,where="C: CallManager")]**
29. **[delegate(SendOps, generics="K",where="K: CustomKernel")]**
30. **[delegate(UpgradeOps, generics="K",where="K: CustomKernel")]**

31. pubstructCustomKernelImpl(pubDefaultKernel);
32. ```
33. fendermint/vm/interpreter/src/fvm/examples/mycustomkernel.rs#L27
34.

1. Implementing all necessary functions for the syscall
2. Implementmy_custom_syscall
3. Here is where we implement our custom syscall:
4.

```

Copy implCustomKernelforCustomKernelImpl where C:CallManager, CustomKernelImpl:Kernel, { fnmy_custom_syscall(&self)->Result { // Here we have access to the Kernel structure and can call // any of its methods, send messages, etc.

// We can also run an external program, link to any rust library // access the network, etc.

// In this example, lets access the file system and return // the number of paths in / letpaths=std::fs::read_dir("/").unwrap(); Ok(paths.count()asu64) } }

```

fendermint/vm/interpreter/src/fvm/examples/mycustomkernel.rs#L42

- Next we need to implement theKernel
- trait for the newCustomKernelImpl
- . You can treat this as boilerplate code and you can just copy it as is:
-

```

Copy implKernelforCustomKernelImpl where C:CallManager, { typeCallManager=C; typeLimiter=asKernel>::Limiter;
```

fninto_inner(self)->(Self::CallManager,BlockRegistry) where Self:Sized, { self.0.into_inner() }

fnnew(mgr:C, blocks:BlockRegistry, caller:ActorID, actor_id:ActorID, method:MethodNum, value_received:TokenAmount, read_only:bool,)->Self { CustomKernelImpl(DefaultKernel::new(mgr, blocks, caller, actor_id, method, value_received, read_only,)) }

fnmachine(&self)->&::Machine{ self.0.machine() }

fnlimiter_mut(&mutself)->&mutSelf::Limiter{ self.0.limiter_mut() }

fngas_available(&self)->Gas{ self.0.gas_available() }

fncharge_gas(&self, name:&str, compute:Gas)->Result { self.0.charge_gas(name, compute) } }

```

fendermint/vm/interpreter/src/fvm/examples/mycustomkernel.rs#L61

1. Link syscalls to the kernel

2. Next we need to implement theSyscallHandler

3. trait for theCustomKernelImpl
4. and link all the syscalls to that kernel. We need to explicitly list each of the syscall traits (ActorOps, SendOps, etc) manually here in addition to theCustomKernel
5. trait. Then inside thelink_syscalls
6. method we plug in the actor invocation to the kernel function that should process that syscall. We can link all the existing syscalls using thelink_syscalls
7. on theDefaultKernel
8. and then link our custom syscall.
9.

```

Copy implSyscallHandlerforCustomKernelImpl where K:CustomKernel +ActorOps +SendOps +UpgradeOps +IpldBlockOps +CryptoOps +DebugOps +EventOps +MessageOps +NetworkOps +RandomnessOps +SelfOps, { fnlink_syscalls(linker:&mutLinker)->anyhow::Result<()> { DefaultKernel:::link_syscalls(linker)?;

linker.link_syscall("my_custom_kernel","my_custom_syscall", my_custom_syscall)?;

Ok(()) } }

```

fendermint/vm/interpreter/src/fvm/examples/mycustomkernel.rs#L112

1. Expose the customized syscall

2. Once this function is linked to a syscall and exposed publicly, we can use this syscall by callingmy_custom_kernel.my_custom_syscall

3.

```

Copy pubfnmy_custom_syscall(context:fvm::syscalls::Context<'_,implCustomKernel>)->Result { context.kernel.my_custom_syscall() }

```

fendermint/vm/interpreter/src/fvm/examples/mycustomkernel.rs#L136

1. Replace existing IPC kernel with new custom kernel

2. Since the customized syscall is implemented in aCustomKernelImpl

3. which extends and implements all the behaviors forDefaultKernel
4. , we can plug it into IPC instead ofDefaultKernel
5. \
6. To use this kernel in fendermint code, replaceDefaultKernel
7. withCustomKernelImpl
8. for theexecutor
```

9. declaration infendermint/vm/interpreter/src/fvm/state/exec.rs
10.

```

Copy

usecrate::fvm::examples::mycustomkernel::CustomKernelImpl;

executor:DefaultExecutor<CustomKernelImpl>>,

```

fendermint/vm/interpreter/src/fvm/state/exec.rs#L86

1. Use syscall in your IPC subnet
2. Now, we are all set to use the custom syscall in the IPC subnet. The custom syscall can be called in IPC actors to utilize the extended feature. For this tutorial, we can create a simple actor to demonstrate how to import and call the custom syscall and then confirm that its working correctly.
3. Let's create acustomsyscall
4. folder inipc/fendermint/actors/
5. and then create a file calledactor.rs
6. in that new folder. Here we want to create a very simple actor, which when invoked (received a message on its Invoke method) will call the new syscall and return its value:
7.

```

Copy fvm_sdk::sys::fvm_syscalls!{ module="my_custom_kernel"; pubfnmy_custom_syscall()->Result; }

pubstructActor; implActor{ fninvoke(rt:&implRuntime)->Result {
rt.validate_immediate_caller_is(std::iter::once(&SYSTEM_ACTOR_ADDR))?;

unsafe{ letvalue=my_custom_syscall().unwrap(); Ok(value) } } }

implActorCodeforActor{ typeMethods=Method;

fnname()->&'staticstr{ CUSTOMSYSCALL_ACTOR_NAME }

actor_dispatch!{ Invoke=>invoke, } }

```

fendermint/actors/customsyscall/src/actor.rs#L14

- Even though this is Rust code, IPC will compile it as a Wasm target and then run the compiled Wasm code inside FVM as an actor. However, we want to share some of the code between Wasm and IPC, such as the actor nameCUSTOMSYSCALL_ACTOR_NAME
- and theInvoke
- method enum. We will define these in a separate file calledshared.rs
- as follows:
-

```

Copy usenum_derive::FromPrimitive;

pubconstCUSTOMSYSCALL_ACTOR_NAME:&str="customsyscall";

# [derive(FromPrimitive)]

# [repr(u64)]

pubenumMethod{ Invoke=frc42_dispatch::method_hash!("Invoke"), }

```

fendermint/actors/customsyscall/src/shared.rs

- We next need to write alib.rs
- file which exports the shared code and only compilesactor.rs
- if we are building the Wasm actor.
-

```

Copy

# [cfg(feature="fil-actor")]

modactor; modshared;

pubuseshared::*;

```

fendermint/actors/customsyscall/src/lib.rs

NOTE: There are several other files you need to change to compile this actor and package it with the other actors that IPC uses. Please refer to the full examplehere for the following other files you need to change:

- fendermint/actors/customsyscall/Cargo.toml
- : The package for your new actor and all its dependencies
- fendermint/actors/Cargo.toml
- : Add your new actor as a Wasm target
- fendermint/actors/build.rs
- : Include your new actor in theACTORS
- array so it will get included in the bundle.
- fendermint/actors/src/manifest.rs
- : Add your new actor in theREQUIRED_ACTORS
- array so we can confirm it was correctly bundled on IPC startup
- fendermint/vm/actor_interface/src/customsyscall.rs
- : A macro which assigns an ID to your new actor and declares constants for accessing it by ID and Address
- fendermint/vm/actor_interface/src/lib.rs
- : export the constants to IPC *

1. Load and deploy actor at genesis

We have so far created a new kernel and syscall, switched IPC to use that kernel and created an actor which calls the new syscall. However, in order to call this actor in IPC, we must load it from the custom_actors_bundle.

- To do this openfendermint/vm/interpreter/src/fvm/genesis.rs
- file and in theinit
- function add our customsyscall actor right after creating thechainmetadata
- actor:
-

```

Copy // Initialize the customsyscall actor which gives an example of calling a custom syscall state .create_custom_actor( fendermint_actor_customsyscall::CUSTOMSYSCALL_ACTOR_NAME, customsyscall::CUSTOMSYSCALL_ACTOR_ID, &EMPTY_ARR, TokenAmount::zero(), None, ) .context("failed to create customsyscall actor")?;

```

fendermint/vm/interpreter/src/fvm/genesis.rs#L251

Your actor has now been deployed and we should be able to send it messages!

1. Invoke the actor

In the last step in this tutorial we will send our customsyscall actor messages which will cause it to run its Invoke method and execute the custom syscall. Here, we will simply call it for every new block height. Go tofendermint/vm/interpreter/src/fvm/exec.rs and inside thebegin function add the following code:

```

Copy letmsg=FvmMessage{ from:system::SYSTEM_ACTOR_ADDR, to:customsyscall::CUSTOMSYSCALL_ACTOR_ADDR, sequence:heightasu64, gas_limit,
```

method_num:fendermint_actor_customsyscall::Method::Invokeasu64, params:Default::default(), value:Default::default(), version:Default::default(), gas_fee_cap:Default::default(), gas_premium:Default::default(), };

let(apply_ret, _)=state.execute_implicit(msg)?;

ifletSome(err)=apply_ret.failure_info { anyhow::bail!("failed to apply customsyscall message: {}", err); }

letval:u64=apply_ret.msg_receipt.return_data.deserialize().unwrap(); println!("customsyscall actor returned: {}", val);

```

fendermint/vm/interpreter/src/fvm/exec.rs#L115

This code sends a message to thecustomsyscall actor and parses it output after it has been executed. We print out the return value from the actor, which will be the return value of our custom syscall.

1. Test your actor

In order to see this working end to end in IPC, you can run one of our integration tests. These tests run IPC in docker containers so make sure to have docker installed on your machine if you are following along.

We must first need to build a new docker container for the fendermint image which will contain all the code you have added so for. To do this run:

```

Copy cdfendermint makedocker-build

```

After the fendermint docker image has been built, you can run one of the integration tests

```

Copy cdfendermint/testing/smoke-test

creates the docker containers

cargomakesetup

runs the integration test

cargomaketest

```

View fendermint logs and see the output generated by calling thecustomsyscall actor in each epoch:

```

Copy dockerps CONTAINERIDIMAGECOMMAND 8da423d8bb1efendermint:latest"fendermint --networ…" ...

```

View the docker logs:

```

Copy dockerlogs8da423d8bb1e ... customsyscallactorreturned:21

```

You can now runcargo make teardown to stop the containers.

Previous Performing transactions in a subnetNext Upgrading a subnet Last updated19 days ago On this page * Pre-requisite knowledge for tutorial: * Instructions * 1. Define the custom syscall * 2. Implementing all necessary functions for the syscall * 3. Link syscalls to the kernel * 4. Expose the customized syscall * 5. Replace existing IPC kernel with new custom kernel * 6. Use syscall in your IPC subnet * 7. Load and deploy actor at genesis * 8. Invoke the actor * 9. Test your actor

Was this helpful? Edit on GitHub
```