{Goals

We're trying to define a global interface API that could be implemented by any container runtime. This Interface purpose is to make easier the implementation of the container spec for VM based container runtimes.
The enhancement provided by the API compared to the OCI CLI will be to prevent upper layers from making any assumptions about PIDs and namespaces in order to seek information and interact with containers.
The unique and only way to interact with sandboxes and containers should be this API.

Containerd recently defined this interface in their v2 shim proto: https://github.com/containerd/containerd/blob/master/runtime/v2/task/shim.proto#L18-L36
We should reuse this interface as it is pretty generic and covers the same goals we're trying to achieve here.

Limitations to containerd approach

The interface and its method are appropriate to the goals we want to achieve here, but there's one limitation that should not be part of this interface. The current API relies on the fact that it has to be implemented as a gRPC server, which is wrong as the implementation choice should be decoupled from the actual API.
The main reason behind this, is to be able to define a generic and widely used API (the same way OCI is used right now), without having to force consumer to choose a technical way to go with it.
Another technical reason we don't want to tie this API with gRPC being the amount of memory consumed by the gRPC protocol and the amount of traffic generated between each component.
This gRPC layer forces runtime implementations to run as a separate process, while they could simply be invoked as a library (Go package).

Proposal

Based on containerd interface, we would remove the gRPC layer and keep the API only. The CRI-O daemon would be the one invoking this API and would be the only process that would spawn all the containers. This means that it would prevent another layer of processes to be spawn because of the gRPC.
CRI-O should define a plugin interface based on the API we're defining here. The goal of the plugin would be to allow container runtimes to provide Golang implementation of this interface.
Now, if we think about not diverging between containerd and CRI-O API, for the sake of the ecosystem, the runtimes implementations for containerd would be a simple gRPC server layer on top of the Golang interface implementation.
This way, CRI-O would keep aligned with the expectation from the ecosystem, and would provide a much more advanced and optimized solution. The global performances would be

better than containerd as the container runtime would be, in a sense, a simple library imported by CRI-O.

CRI-O codebase will need to be modified and updated in order to support this.

Technical details

The first thing that comes to mind is to ensure the current codepath (preferred for runc runtime) should not be affected by any of those coming changes since we still want the same support and performances for the OCI CLI container runtimes. In order to honor this, the current set of configuration options regarding trusted runtimes and workloads should be extended with a new option **version** used to select the appropriate code path. By default, if the version is not provided (empty), it should always use the default code path which is the OCI CLI one.
But in case the version is provided as **v2** (to be defined) for instance, the code path of the new interface should be chosen.
One thing to notice, because **runc** will not support this **v2** version, if the flags about runtimes and workloads suggest a trusted runtime to be picked, the version should not matter and **runc** should be picked, using the default code path.

Second thing, we have over-simplified the way this interface would be integrated with CRI-O from the description above. Obviously, we cannot simply import the Golang package implemented by each VM based runtime that will implement this API, it purely impossible. Just to clarify, let's say we have both Kata Containers and Gvisor providing an implementation of this API, the CRI-O code will not import both of them since we don't know which one should be used at build time.
Instead, and to give more technical details about the solution proposed here, we should create some plugins implementing this library. Each implementation would be compiled as a shared library, and in parallel CRI-O would be compiled relying on the declaration of this API. CRI-O would need to be modified so that from a path (**../../kata_containers_v2.so** being the path to the shared library that the user wants to use) provided through the config file **/etc/crio/crio.conf**, it would load the library at runtime. This would allow CRI-O to be previously built with no clue about the implementation being used until the moment it is actually running and loading the associated shared library.
This solution is the only one that can be considered if we still want to provide a proper layer of isolation between CRI-O and each implementation of the interface API proposed here.

Third point, a pretty important one, the definition of the interface. Even if the interface proposed by containerd-shim-v2 seems good, we have decided to go the other way by thinking about each function that might be needed by this interface, as a replacement for OCI.
The plan being that if we end up with some delta with the containerd-shim-v2 interface, we'll try to push this delta to their interface as we think this will make sense in order to keep those interfaces identical (don't forget we don't want to diverge).

Opens

One concern might be about the interface forcing the implementation to grab information from disk every time a call to one of the function is done. What I am trying to say is that compared to the gRPC solution including the interface, the implementation will be very different here since the only long living daemon will be CRI-O. And for every function call, CRI-O will not have the internal structures used by the implementation, forcing the implementation itself to reload those structures from disk.

Would it be possible to return an anonymous interface through each call of the API, so that CRI-O could re-inject this interface(user data basically) through each following calls?
TODO: Add details about why this would be needed

We can pick oci.go as a starting point and then modify it to add what is missing and make it better:

```
type Runtime interface {
   CreateContainer(c *Container) error
   StartContainer(c *Container) error
   StopContainer(ctx context.Context, c *Container, timeout int64) error
   ExecContainer(c *Container, command []string, timeout int64, sync bool) (resp
*ExecResponse, err error)
   UpdateContainer(c *Container, res *rspec.LinuxResources) error
   DeleteContainer(c *Container) error
   WaitContainerStateStopped(c *Container) error
   UpdateContainerStatus(c *Container) error
   SetContainerStatus(c *Container, state) error
   ContainerStatus(c *Container) *ContainerState
   PauseContainer(c *Container) error
   ResumeContainer(c *Container) error
   PortForward(c *Container, port int32) error
   Stats(c *Container) StatsInfo
}
```