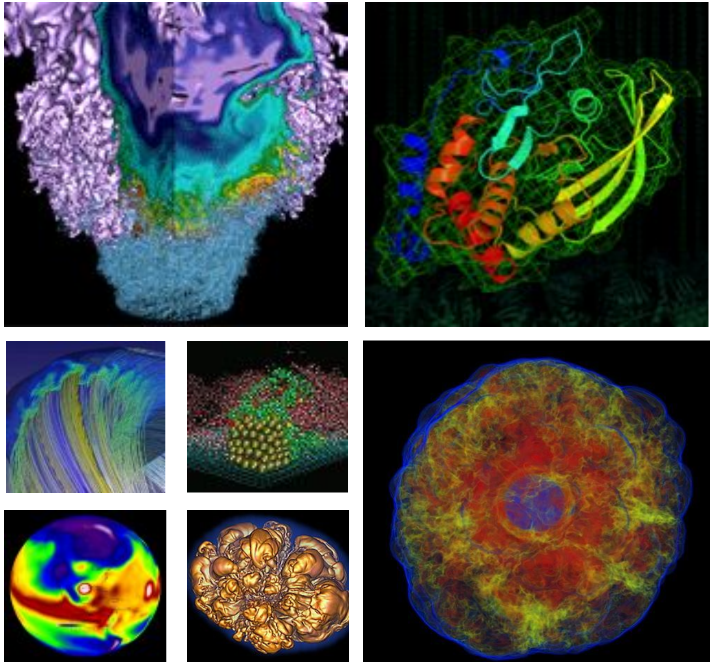
Job Container plugin for managing node local namespaces



NERSC

Aditi Gaur

NERSC

Slurm user group 2019, Utah

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB

Job container Plugin- Use Case

- NERSC Cori System extended
  - GPU Racks- Each GPU node consists of local SSD’s
  - Big Memory Nodes- For bioinformatics pipelines, 20 nodes, 1.5 TB
- Users want private scratch space
- A private scratch space, configurable, and on-demand
- Each job should allocate a local /tmp and private /dev/shm to the job
  - Clean up /tmp on teardown
  - Make sure shared jobs cannot interfere with each others allocations

Job container Vs Spank

- Spank provides pluggable routines before, during and after job launch, and the same for teardown. But implementing functionality in spank means other spank plugins can only see the namespace created if they run after the namespace spank plugin.

- Job container creation runs before all spank prologs, and teardown happens after all spank epilogs. Hence spank plugins are attached to the namespace created by job container plugin

- A corresponding functionality in spank, would have to rely on ordering of spank plugins, if other spank plugins want to use the namespace or expect to be in namespace.

- Job container plugin provides the required wire up for adding other namespaces as well. This meets the need of a generalized infrastructure of handling namespaces in slurm
- More maintainable than spank

Concepts- Linux Namespaces

- Encapsulate a global system resource
  - Processes inside, have an isolated view of the resource
  - Processes view resource as exclusive

- 7 kinds of Namespaces-
  - Mount , `CLONE_NEWNS` → similar to chroot jails but more flexible and secure
  - PID , `CLONE_NEWPID` → isolates the process table
  - UTS, `CLONE_NEWUTS` → isolates nodename and domainame
  - IPC , `CLONE_NEWIPC` → isolates IPC resources
  - Network , `CLONE_NEWNET` → isolates networking resources such as IP tables, IP addresses
  - User, `CLONE_NEWUSER` → latest addition to the kernel, Isolates UID and GID
  - Cgroups, `CLONE_NEWCGROUPS`

Concepts -Linux Namespaces (2)

- The proc filesystem

```
$ ls -l /proc/self/ns
      total 0
      lrwxrwxrwx. 1 mtk mtk 0 Apr 28 12:46 cgroup -> cgroup:[4026531835]
      lrwxrwxrwx. 1 mtk mtk 0 Apr 28 12:46 ipc -> ipc:[4026531839]
      lrwxrwxrwx. 1 mtk mtk 0 Apr 28 12:46 mnt -> mnt:[4026531840]
      lrwxrwxrwx. 1 mtk mtk 0 Apr 28 12:46 net -> net:[4026531969]
      lrwxrwxrwx. 1 mtk mtk 0 Apr 28 12:46 pid -> pid:[4026531836]
      lrwxrwxrwx. 1 mtk mtk 0 Apr 28 12:46 pid_for_children -> pid:[4026531834]
      lrwxrwxrwx. 1 mtk mtk 0 Apr 28 12:46 user -> user:[4026531837]
      lrwxrwxrwx. 1 mtk mtk 0 Apr 28 12:46 uts -> uts:[4026531838]
```

- If 2 process have same namespace- the symbolic links would point to same inode

Concepts- Linux Namespaces (3)

System calls -

- Clone
  - Forks a new process, isolates the requested resources in the child process from the parent

```
/* GLIBC wrapper is different from clone system call*/
```

- Unshare
  - Does not call fork, but stops sharing the requested resource from parent. Future events on resource are only visible to the calling process.

```
/* Fork + unshare, provides functionality similar to clone().
Fork does internally call clone but without namespace flags*/
```

- setns
  - Enter a namespace that is alive
  - Requires an open fd to the proc filesystem

Concepts- Linux Namespaces (3)

- A word about using Clone vs Unshare
- Clone creates a new process, and applies namespace semantics to it.
- Unshare “un shares” the requested namespace of the calling process from the parent
    - It “leaves” the namespace of the parent.
    - Does not fork new processs
    - Its easier to create a persistent namespace using fork + unshare

```
/* Create child that has its own UTS
namespace, child commences execution in
childFunc() */

pid = clone(childFunc, stackTop,
      CLONE_NEWUTS | SIGCHLD, argv[1]);
…
}

Static int childFunc(void *arg) {
      /* cloned child in UTS namespace
starts here */
… }
```

```
cpid = fork( );

If (cpid ==0) {
      unshare(CLONE_NEWNS|CLONE_NEWUSER;
      /*child has left mount and user
namespace*/
      …
}

/*parent*/

continue;
```

Creating Persistent Namespace

- Namespaces created can only be kept alive by keeping a process alive inside it.

- To avoid unnecessary bottleneck- We use bind mounting to keep namespace alive

- The bind mount keeps the namespace alive because- We can open an fd to `/proc/$PID/ns/mnt`, to enter the namespace in `setns( )` call

    - This makes is possible to have a filesystem of open namespaces

Creating persistent namespace

Example code:

```
/* parent*/
    cpid = fork( );
    if (pid == 0) {
        /*child*/
        unshare(CLONE_NEWNS);
        …  /* mechanism to wait while parent bind mounts*/
    }
    /*parent*/
    mount("/proc/cpid/ns/mnt", path, NULL, MS_BIND, NULL);
        …
    }
```

Job Container - API

- In slurm.conf use ‘`JobContainerType`’ to use job container

```
JobContainerType=job_container/tmpfs
```

- Currently 2 job container plugins exist
  - cncu - cray only plugin (compute node clean up)
  - none
- Job Container Plugin initialized at the start of slurm Daemons
  - container_p_restore( ) → global initialization can be put here
- container_p_create( ) → create new container
- container_p_delete( ) → called after spank epilogs, destroy the container
- container_p_join() → Called before slurmstepd forks any tasks for the job, add pid to the job container

Job container - Namespace.conf

- Namespace.conf provides ability to configure options
- Currently supports
  - `Basepath = Path that job container plugin should Use for constructing creating job's /tmp.`
  - `NodeName = For each NodeName, can have a different configuration`
  - `InitScript = optional initscript, for running any initialization`

Namespace.conf Example

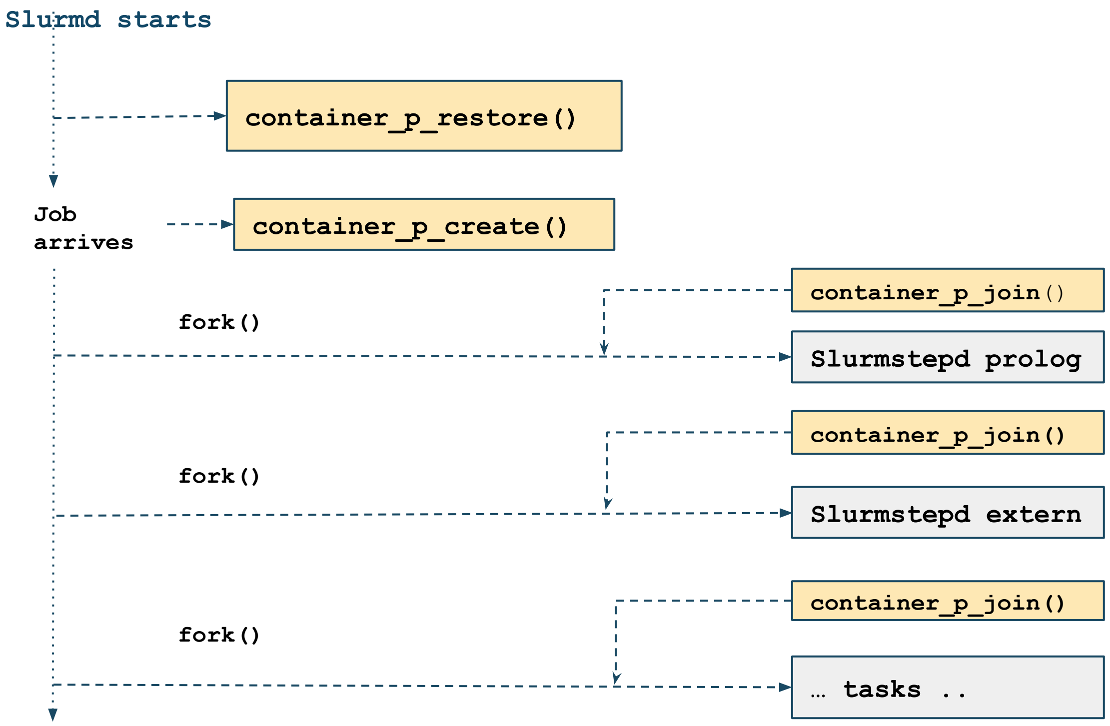
How we use at NERSC:

```
NodeName=cgpu[01-18] BasePath=/var/opt/nersc/nvme
NodeName=exvivo[001-020] BasePath=/var/opt/nersc/nvme
NodeName=cori[01-08,10-21] BasePath=/var/opt/cray/persistent
```

Possible Use Cases & Future work

- Interesting use case to support XFS (and/or BTRFS) quotas. When user gets a namespace assigned, they also get assigned relevant disk space, which is private scratch for their job.
- User can get added into multiple namespaces. One example would be- users only get to see filesystems they have requested access to, with different permissions such as read only that can be controlled via mounting.
- Easy wire up to support more namespaces than just mount. Maybe Cgroups.
- This provides a better way of implementing PID namespaces
- Plans to upstream for 20.02

Job Container API - Job Launch



Slurmd starts

container_p_restore()

Job arrives

container_p_create()

fork()

container_p_join()

Slurmstepd prolog

fork()

container_p_join()

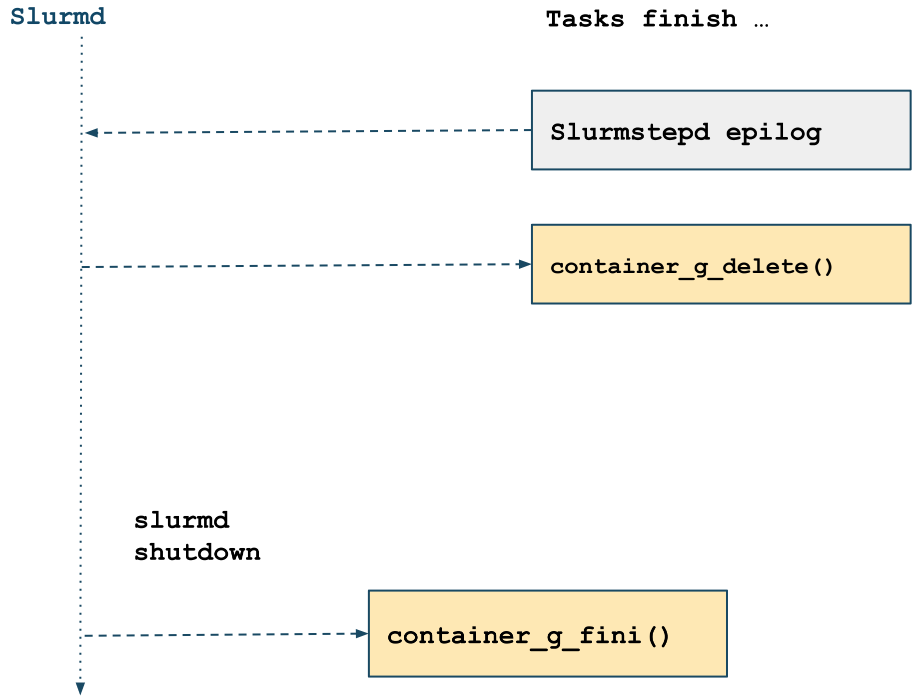
Slurmstepd extern

fork()

container_p_join()

… tasks ..

Job Container API: Teardown



Slurmd

Tasks finish …

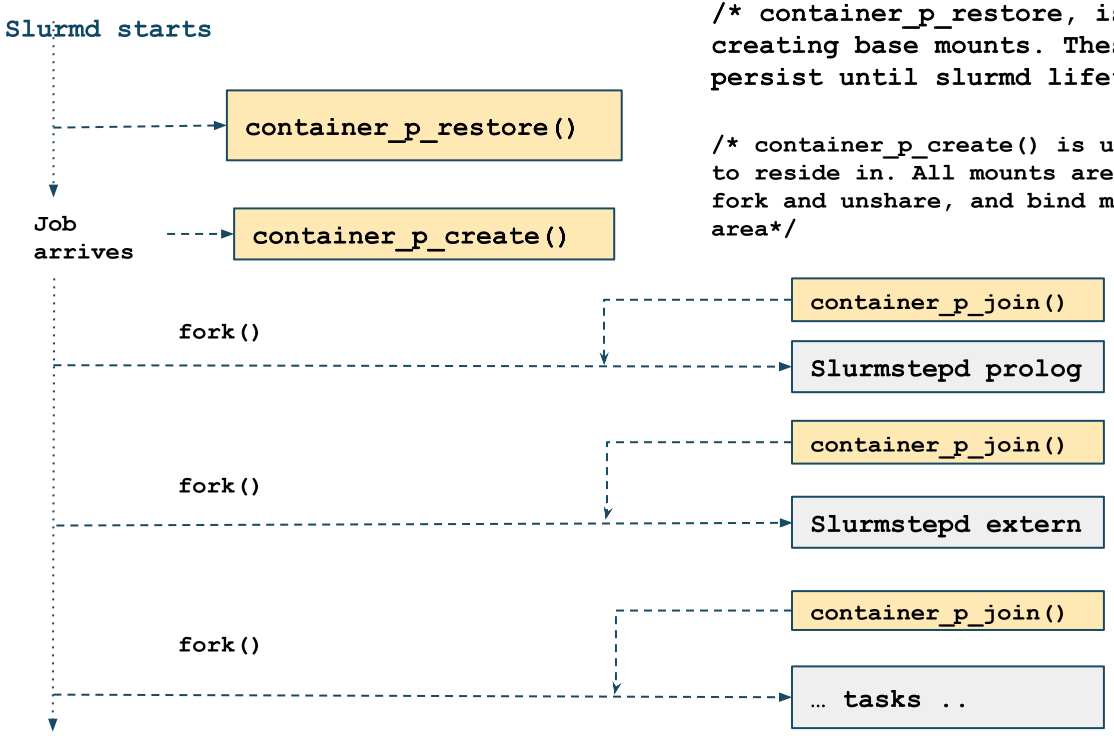
Slurmstepd epilog

container_g_delete()

slurmd
shutdown

container_g_fini()

Job Container/tmpfs - Job Launch

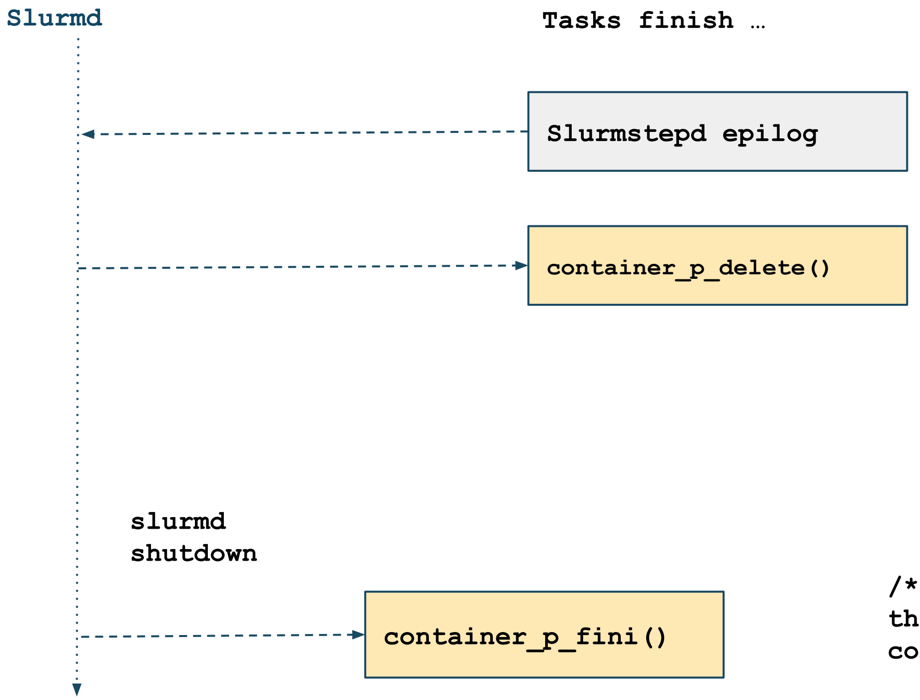


/* container_p_restore, is used for creating base mounts. These mounts persist until slurmd lifetime */

/* container_p_create() is used for creating the namespace, job is to reside in. All mounts are created for the job in this call. We fork and unshare, and bind mount the namespace to a persistent area*/

/* container_p_join() calls setns using the open fd of the bind mount created above. This makes the process join the namespace, and see namespace specific resources */

Job Container/tmpfs: Teardown



```
/* container_p_delete destroys the container
by unmounting the bind mount. It also
deletes all the files created by the user,
in the scratch space */
```

```
/* container_p_fini cleans up
the base mounts created in
container_p_restore() */
```

Job Container/tmpfs

Multiple jobs on a Node:

```
root@linux_vb:/usr/local/etc# findmnt
TARGET                                SOURCE                 FSTYPE     OPTIONS


├─/storage                            /dev/sdb               xfs        rw,relatime,attr2,inode64,usrquota
│ └─/storage                          /dev/sdb               xfs        rw,relatime,attr2,inode64,usrquota
│   ├─/storage/2/.ns                  nsfs[mnt:[4026532307]] nsfs       rw
│   ├─/storage/3/.ns                  nsfs[mnt:[4026532313]] nsfs       rw
│   └─/storage/4/.ns                  nsfs[mnt:[4026532314]] nsfs       rw
```

Job Container/tmpfs

What jobs see (inside namespace):

```
parallels@linux_vb:~$ findmnt

└─/tmp                                /dev/sdb[/2/.2]     xfs        rw,relatime,attr2,inode64,usrquota
```

Conclusion

- Job container provides a more mature wire-up to support namespaces in slurm than spank
- Provides, better encapsulation for jobs
- Namespace.conf provides configurable way to support the job container API
  - Additional options in namespace.conf can easily be added to support more functionality

Thank You

Job Container API - Job Startup

→ slurmd starts

→ `container_g_init( ) /* this function gets called in many different contexts*/`

→ `container_g_restore( ) /* useful for node-level initialization, only slurmd context*/`

→ job_arrives (`_rpc_prolog( )`)

  → `container_g_create( ) /* Now we are doing job specific initialization*/`

  → creates slurmstepd with option to only run prolog, slurmstepd is also added to the container via `container_g_join()`

    → `container_g_init( ) /*stepd context here, for spank prolog */`

    → spank prolog for all spanks

      → on each `fork() /* before execve */`

        → `container_g_join() /* add the spank pid to the container*/`

        → `container_g_add_cont() /*add proctrack container to job container*/`

Back to slurmd ←

  → `_forkandexecslurmstepd( )` to run job tasks

    → `container_g_init( ) /* stepd context here */`

    `/* all slurm stepd forks call container_g_join and container_g_add_cont*/`

Job Container API - Job teardown

Job_ends (slurmstepd)

→ call spank epilogs

```
container_g_delete( ) /*remove the job container and de-allocate any memory
                        Perform any node-level teardown
                        NOTE: This runs AFTER spank epilogs*/
```

Back to slurmd ← slurmstepd_exits

When slurmd exits:

```
    → job_container_fini( ) /* slurmd context, now remove any global
                                         Initializations */
```

Job Container - TMPFS - Job Startup

slurmd starts

`container_g_init( ) /*No - op*/`

`container_g_restore( ) /*Create Base Namespace */`

Job_arrives (`_rpc_prolog( )`**)**

- → `container_g_create( ) /* create project level directories, set permissions, unshare mount namespace, mount a private /tmp, /dev/shm inside it. Make namespace persistent by bind mounting*/`

`/* All forks after this call container_g_join*/`

- → creates slurmstepd with option to only run prolog
  - → `container_g_init( ) /*no - op for our plugin */`
  - → spank prolog for all spanks
    - → on each `fork() /* before execve */`
      - → `container_g_join() /*nsenter() into our namespace created*/`

Back to slurmd ←

- → `_forkandexecslurmstepd( )` to run job tasks. Every fork before execve will call
  - → `container_g_join( ) /* nsenter all job processes */`

Job Container TMPFS - Job teardown

Job_ends (slurmstepd)

→ call spank epilogs

```
container_g_delete( ) /* teardown the namespace
                         Go into the directory, and  clean up all the files.
                         NOTE: This runs AFTER spank epilogs*/
```

Back to slurmd ← slurmstepd_exits

When slurmd exits:

```
    → job_container_fini( ) /*  teardown base namespace here */
```