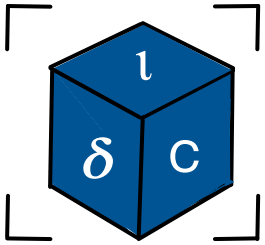
Resource Virtualization with Containers

Tanu Malik

School of Computing, DePaul University

Visiting Faculty, CSE, IIT, Delhi



Namespaces

- Is both an isolation and sharing mechanism
  - A process within a namespace has its **own isolated instance** of the resource, which is shared with other processes within the same namespace

- Note namespaces have no names and are identified by numbers.

Namespace management

- Creating a new namespace
  - clone(function, stack, CLONE_NEW*, args)
    - creates a new process and a new namespace; the new process is attached to the new namespace.
- Joining an existing namespace
  - setns(fd, nstype)
    - calling process to join an existing namespace specified by the file descriptor and nstype.
- Redefining namespace
  - unshare(flags)
    - Disassociating parts of process execution contexts
- Discovering namespace relationships
  - ioctl(fd, request)
    - discovery of namespace relationships

Seven types of namespaces

These namespaces isolate resources

- PID NS: isolates pid resource
- Mount NS: isolates mount point resource
- Network NS: isolate of network system interface and device resource
  - network interfaces, routing tables, DNS lookup servers, IP addresses, subnets
- IPC NS: isolation of IPC and POSIX message queues resource
- UTS NS: isolation of hostname resource
  - Exists for historical reasons
- User NS: isolates owner of a process respurce
- Cgroup: isolated view of the resource usage

Namespaces have numbers

- Each type of namespace is identified by an inode (unique)
- six entries (inodes) added to /proc/<pid>/ns/
- two processes are in the same namespace if they see the same inode for equivalent namespace types (mnt, net, user, ...)

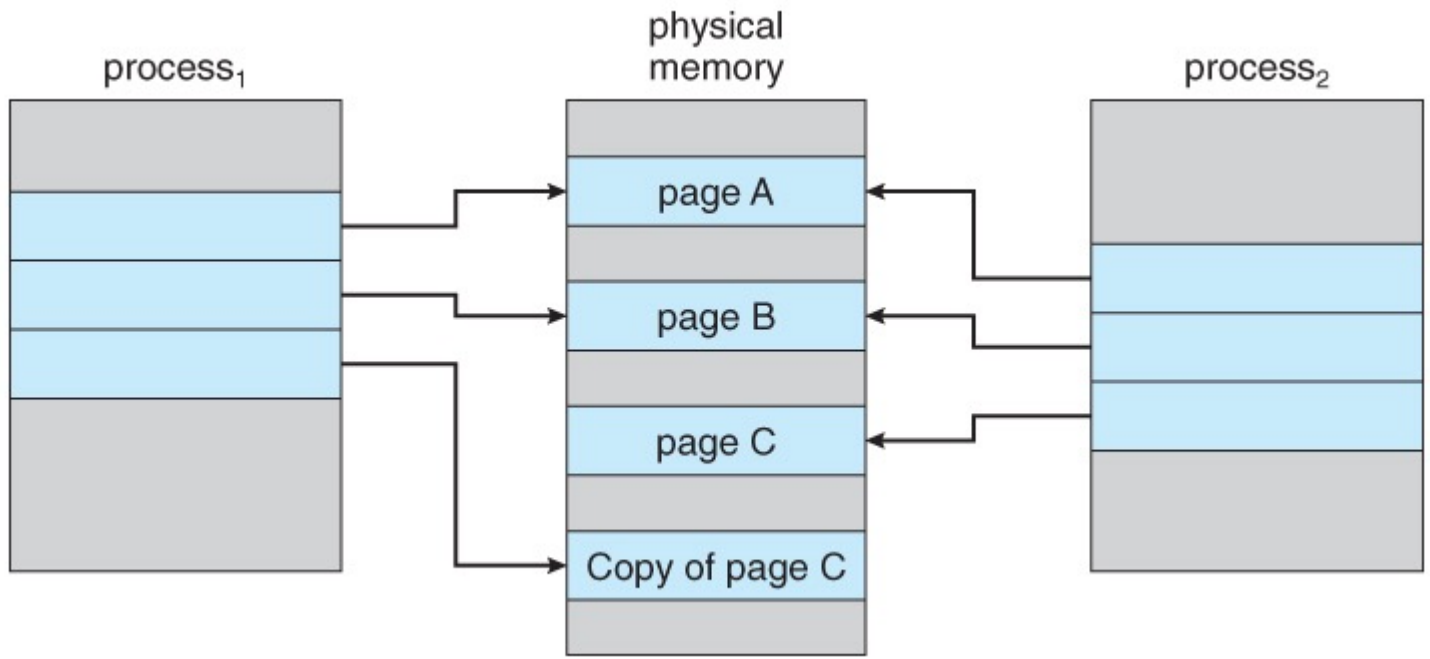
- lsns
- `ls –l /proc/$$/ns`
- `ls –l /proc/<pid>/ns`

Resources not yet isolated

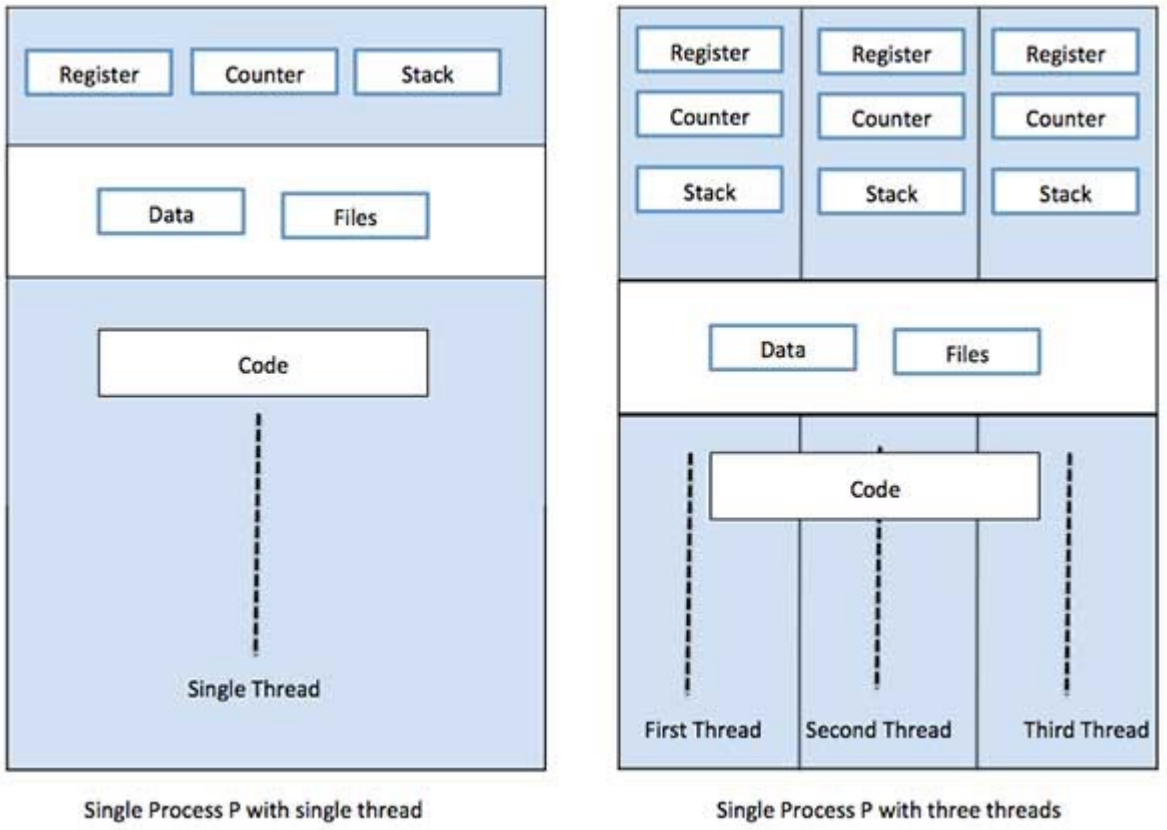
- time namespace.
- secure keys
- device namespace

Creating a new namespace

- The *clone* system call helps to create a new child process in a manner similar to fork, but provides precise control over execution context shared between caller and callee
- Flags in the syscall help to further control what is shared
  - E.g. share virtual address space, the table of file descriptors, and the table of signal handlers.
- Flags also control if new child process in the same namespace as the parent


process1
physical memory
process2
page A
page B
page C
Copy of page C

Threads



Difference between creating process and thread in Linux is just different arguments to clone() syscall!

Process Vs Threads

Processes are heavyweight operations	Threads are lighter weight operations
Each process has its own memory space	Threads use the memory of the process they belong to
Inter-process communication is slow as processes have different memory addresses	Inter-thread communication can be faster than inter-process communication because threads of the same process share memory with the process they belong to
Context switching between processes is more expensive	Context switching between threads of the same process is less expensive (Use same address space)
Processes don’t share memory with other processes	Threads share memory with other threads of the same process

clone()

int clone(int (*child_func)(void *), void *child_stack, int flags, void *arg);

- First argument is a pointer to a function which returns an integer.
  - Cloned child does not proceed from next instruction in parent but from the first instruction specified in *child_func*

- *child_stack* specifies the location of the memory stack used by the child process.
  - Note, size of stack remains unknown to the kernel as passed memory is just a pointer
  - The programmer must decide how much space is required for the operations the child process will perform.
- *arg* is a pointer to the arguments that will pass to the function that the child process will execute.
- *flags* control two things:

1. child's *termination signal*, which is the signal to be sent to the parent when the child terminates.
2. execution context sharing between the parent and the child
   - About twenty different CLONE_* flags that control clone() shared operation, six of which create namespaces.
   - The new child process is in the flag specified namespace.

- Returns processid of child on return or -1 on error.
- Cloned child terminates at exit() or child_func *return.* Parent process waits in the usual manner.

Clone flags

- CLONE_VM - share memory
  - If the CLONE_VM flag is set, then the parent and child share the same virtual memory pages (hallmark of threads)
  - Updates to memory or calls to *mmap()* or *munmap()* by either process will be visible to the other process.
  - If the CLONE_VM flag is not set, then the child receives a copy of the parent's virtual memory (as with *fork()*).

- CLONE_FILES - share file descriptors
  - If the CLONE_FILES flag is specified, the parent and the child share the same table of open file descriptors.
  - Updates to file descriptor allocation or deallocation (*open()*, *close()*, *dup()*, *pipe()*, *socket()*, and so on) in either process will be visible in the other process.
  - If the CLONE_FILES flag is not set, then the file descriptor table is not shared, and the child gets a copy of the parent's table at the time of the *clone()* call.

Flags

- CLONE_FS
  - If the CLONE_FS flag is specified, then the parent and the child share file system related information: umask, root directory, and current working directory.
  - calls to *umask()*, *chdir()*, or *chroot()* in either process will affect the other process.
  - If the CLONE_FS flag is not set, then the parent and child have separate copies of this information (as with *fork()*).

- CLONE_SIGHAND
  - Parent and child share signal dispositions

Flags for namespaces

CLONE_NEWNS	2.4.19	CAP_SYS_ADMIN
CLONE_NEWUTS	2.6.19	CAP_SYS_ADMIN
CLONE_NEWIPC	2.6.19	CAP_SYS_ADMIN
CLONE_NEWPID	2.6.24	CAP_SYS_ADMIN
CLONE_NEWNET	2.6.29	CAP_SYS_ADMIN
CLONE_NEWUSER	3.8	No capability is required

Clone examples

- basic_clone
- e.c
- http://man7.org/tlpi/code/online/dist/procexec/demo_clone.c.html

Joining an existing namespace: setns()

- The setns() system call allows the calling process to join an existing namespace identified by *fd* and *nstype*:
  - int setns(int fd, int nstype);
- Note process is always in some namespace
  - setns() disassociates the calling process from one instance of a particular namespace type and reassociates the process with another instance of the same namespace type.

setns()

- int setns(int fd, int nstype);

- *fd* argument specifies the namespace to join
  - it is a file descriptor that refers to one of the symbolic links in a /proc/*PID*/ns directory.
  - file descriptor can be obtained either by opening one of those symbolic links directly or by **opening a file that was bind mounted to one of the links.**

- The *nstype* argument allows the caller to check the type of namespace that fd refers to.
  - If this argument is specified as zero, no check is performed.
    - This can be useful if the caller already knows the namespace type, or does not care about the type.
  - *nstype* = CLONE_NEW*
  - verifies that *fd* is a file descriptor for the corresponding namespace type.
    - Useful if the caller was passed the file descriptor via a UNIX domain socket and needs to verify what type of namespace it refers to.

Unsharing execution contexts: unshare()

int unshare(int flags);

- Functionality similar to clone() but operates on the caller instead of the callee
- It creates the new namespaces specified by the CLONE_NEW* bits in its flags argument and makes the caller a member of the namespaces.
- Main purpose of unshare() is to isolate namespace (and other) side effects without having to create a new process or thread (as is done by clone()).

Example of unshare

- `clone(..., CLONE_NEWXXX, ....);`

is roughly equivalent, in namespace terms, to the sequence:

```
if (fork() == 0)
   unshare(CLONE_NEWXXX);
```

- `Unshare is also available as a command`
  - unshare [options] program [arguments]
  - options are command-line flags that specify the namespaces to unshare before executing program with the specified arguments.

Namespace termination

- A namespace remains open as long there is an open resource
- Non-existence of a process does not imply non-existence of namespace

Six namespaces

- UTS NS: uts (unix timesharing - domain name, etc) CLONE_NEWUTS
- PID NS: pid (processes) CLONE_NEWPID

- Mount NS: mnt (mount points, filesystems) CLONE_NEWNS
- User NS: user (UIDs) CLONE_NEWUSER

- Network NS: net (network stack) CLONE_NEWNET
- IPC NS: ipc (System V IPC) CLONE_NEWIPC

Previlige for namespaces

CLONE_NEWNS	2.4.19	CAP_SYS_ADMIN
CLONE_NEWUTS	2.6.19	CAP_SYS_ADMIN
CLONE_NEWIPC	2.6.19	CAP_SYS_ADMIN
CLONE_NEWPID	2.6.24	CAP_SYS_ADMIN
CLONE_NEWNET	2.6.29	CAP_SYS_ADMIN
CLONE_NEWUSER	3.8	No capability is required

Namespaces with clone

- CLONE_NEW* bits is specified in the call, then a new namespace of the corresponding type is created, and the new process is made a member of that namespace; multiple CLONE_NEW* bits can be specified in flags.

Hostname Namespace

- isolate two system identifiers—nodename and domainname—returned by the uname() system call
  - the names are set using the sethostname() and setdomainname() system calls.
- In the context of containers, the UTS namespaces feature allows each container to have its own hostname and NIS domain name.
- UTS namespaces are useful for initialization and configuration scripts that tailor their actions based on these names.
- The term "UTS" derives from the name of the structure passed to the uname() system call: struct utsname. The name of that structure in turn derives from "UNIX Time-sharing System".

UTS namespace with CLONE_NEWUTS

```
child_pid = clone(childFunc, child_stack +
STACK_SIZE, CLONE_NEWUTS | SIGCHLD, argv[1]);
printf("PID of child created by clone() is
%ld\n", (long) child_pid);
```

- As with most other namespaces (user namespaces are the exception), creating a UTS namespace requires privilege (specifically, CAP_SYS_ADMIN)

Example 1

- Demo_uts_namespace.c
  - Show two different namespaces
  - Show namespace does not exist
- Force existence
  - # **touch ~/uts** # Create mount point
  - # **mount --bind /proc/<CHILD_PID>/ns/uts ~/uts**
- setns.c
  - **./setns ~/uts /bin/bash**
  - # hostname
  - # ls –l **/proc/$$/ns/**

Example 2

- Unshare.c
- **# echo $$**
  - # Show PID of shell
- **# cat /proc/<PID>/mounts | grep mq**
  - # Show one of the mounts in namespace
- **# sudo ls -l /proc/<PID>/ns/**
  - # Show mount namespace ID
- **# ./unshare -m /bin/bash**
  - # Start new shell in separate mount namespace
- **# sudo ls -l /proc/$$/ns/**
  - # Show mount namespace ID