COMP3301/COMP7308 Assignment 2

- School of Information Technology and Electrical Engineering
- The University of Queensland
- Semester Two, 2015
- Due: 8pm Friday 18 September 2015
- Revision: $Revision: 82 $

1 Process Isolation for Containers

This assignment asks you to implement a portion of the functionality necessary to provide "containers" on OpenBSD.

Container technologies aim to provide virtualisation at the kernel level instead of at the hardware level. This is implemented by extending a kernel to partition and isolate certain services to prevent processes in a container from interacting with processes in another container. Such isolation may require limiting the visibility of processes and file descriptors in the system, creating independent users and views of the file system, virtualising the network stack, and guaranteeing access to resources.

Several container technologies exist such as Docker, Solaris Zones, FreeBSD jails, and AIX WPARs. The tasks in this assignment will be loosely modelled on the design of Solaris Zones as documented in PSARC/2002/174. The aim will be to implement isolation of processes in OpenBSD.

This is an individual assignment. You should feel free to discuss aspects of C programming and the assignment specification with fellow students. You should not actively help (or seek help from) other students with the actual coding of your assignment solution. It is cheating to look at another student's code and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code may be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you. If you're having trouble, seek help from a member of the teaching staff. Don't be tempted to copy another student's code. You should read and understand the statements on student misconduct in the course profile and on the school web-site: http://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism

2 Specification

You will add basic support for process isolation to the OpenBSD kernel and user land by implementing a simplified zones infrastructure modelled on the Solaris specification and implementation.

Process isolation is the prevention of a process in one zone being able to see or signal a process running in another zone.

The exception to this isolation will be the "global" zone, which can view and signal all processes in the system including those running in other zones. The ability to create, destroy, and enter zones will also be limited to the root user in the global zone.

2.1 Code Style

Your code is to be written according to OpenBSD's style guide, as per the `style(9)` man page.

2.2 Compilation

Your modifications and additions to the OpenBSD system must be integrated into the existing build infrastructure. The system will be rebuilt with your supplied code by following the process in the `release(8)` man page.

2.3 Types and Limits

The following types and limits should be made available to the system (both kernel and userland).

2.3.1 zoneid_t

Zones will be uniquely identified in the running system by a numerical identifier. The system headers should provide the following typedef to expose this identifier:

```
typedef int zoneid_t;
```

2.3.2 MAXZONENAMELEN

Zones will also be uniquely identified by a name in the running system. The system headers should expose a `MAXZONENAMELEN` macro that states what the longest name for a zone may be including a terminating `NUL` character.

2.3.3 MAXZONES

The system should be limited to only providing up to 1024 zones at a time.

2.3.4 MAXZONEIDS

While the system is limited to running 1024 zones, the limit on identifiers is higher to avoid rapid reuse.

2.3.5 struct process

`struct process` represents the kernels state relating to a running program. This type should be extended to record which zone the process is running in.

2.3.6 struct kinfo_proc

`struct kinfo_proc` is a serialisation of the kernels process and proc structures for userland to consume.

It should be extended to include the id of the zone that it is running in.

Care must be taken to avoid ABI compatibility issues when modifying `struct kinfo_proc` by only adding members at the end of the structure.

2.4 Interfaces

The following system calls need to be added to the OpenBSD system

2.4.1 zone_create

```
zoneid_t        zone_create(const char *zonename);
```

zone_create should create a new zone id for use in the system.

On success it should return the zone id that was created.

On failure it should return -1 and set errno accordingly:

EPERM the current program is not in the global zone
EPERM the current user is not root
EEXIST a zone with the specified name already exists
ERANGE too many zones are currently running
EFAULT zonename points to a bad address
ENAMETOOLONG the name of the zone exceeds MAXZONENAMELEN

2.4.2 zone_destroy

```
int                 zone_destroy(zoneid_t z);
```

zone_destroy should delete the specified zone instance.

On success it should return 0.

On failure it should return -1 and set errno accordingly:

EPERM the current program is not in the global zone
EPERM the current user is not root
ESRCH the specified zone does not exist
EBUSY the specified zone is still in use, ie, a process is still running in the zone

2.4.3 zone_enter

```
int                 zone_enter(zoneid_t z);
```

zone_enter moves the current process into the zone.

On success it should return 0.

On failure it should return -1 and set errno accordingly:

EPERM the current program is not in the global zone
EPERM the current user is not root
ESRCH the specified zone does not exist

2.4.4 zone_list

```
int                 zone_list(zoneid_t *zs, size_t *nzs);
```

In the global zone `zone_list` will provide the list of zones in the running system as an array of zoneid_ts. If run in a non-global zone, the list will only contain the current zone.

The value at `nzs` will refer to the number of array entries in `zs` on input.

On success it should return 0 and the value at `nzs` will be set to the number of zones listed in `zs`.

On failure it should return -1 and set errno accordingly:

EFAULT `zs` or `nzs` point to a bad address
ERANGE if the number at `nzs` is less than the number of running zones in the system

2.4.5 `zone_name`

```
int                 zone_name(zoneid_t z, char *name, size_t namelen);
```

`zone_name` provides the name of the zone identified by `z`. If run in a non-global zone, only the current zone may be specified.

On success it will return 0.

On failure it should return -1 and set errno accordingly:

ESRCH The specified zone does not exist
ESRCH The specified zone is not visible in a non-global zone
EFAULT `name` refers to a bad memory address
ENAMETOOLONG The requested name is longer than `namelen` bytes.

2.4.6 fork(2)

When a process forks, the child must inherit the zone it is running in from its parent.

The only way for a process to change zones is via the `zone_enter` syscall, which is limited to root processes in the global zone.

2.4.7 kill(2)

The kernel signalling code should be modified to provide the following semantics:

- If any user in a non-global zone tries to signal any process in another zone, it should fail with ESRCH.
- If a non-root user in the global zone signals a process in another zone, it should fail with EPERM.
- root in the global zone may signal any process in any zone
- Users within a zone should get normal signalling semantics

2.4.8 sysctl(3)

The kernel side of sysctl should modify its handling of CTL_KERN KERN_PROC to filter results.

- the global zone does not get a filtered list of processes
- non-global zones will get a list of processes that exist in their current zone
- `struct kinfo_proc` structure must be modified to include a ps_zoneid field which identifies the zone the process is running in

2.5 Userland Libraries

2.5.1 `libc`

`libc` should be updated to provide stubs for the system calls described above. The system should also provide a `zones.h` file for installation in `/usr/include` that prototypes the system call stubs.

Programs needing to interact with the kernels zone infrastructure can `#include <zones.h>` and link to `libc`.

A manual page is **NOT** necessary for the system call stubs.

2.6 Userland Programs

To administer the zones subsystem, some userland utilities should be modified and implemented.

When appropriate, programs should be modified to accept the following options:

-z zone Limit the scope of the command to the specified zone. The zone may be specified by name, or by numeric id.

-Z The name of the zone should be added to the programs output.

All changes should be documented in the relevant manual pages.

The following changes should be implemented:

2.6.1 ps(1)

- add the -z option

When a zone is specified, the list of processes displayed by ps will be limited to those processes running in the specified zone.

- add the -Z option

-Z should cause the zones name to be prepended to the columns that are output by ps(1). The column should be a maximum of 8 characters wide, with its value right aligned. If a zone name exceeds 8 characters, it should be truncated to 7 characters and suffixed with an asterisk (*).

Additionally, “zone” may be specified as a column in custom column format specifiers.

2.6.2 pgrep, pkill

- add the -z flag

pgrep and pkill will only match on processes that are running in the specified zone.

2.6.3 zone(8)

zone(8) will be a new program and should be installed under `/usr/sbin`.

The required functionality is described by the usage shown below:

```
usage: zone create zonename
zone destroy zonename|zoneid
zone list
zone exec zonename|zoneid program [arg ...]
```

2.7 Recommendations

2.7.1 APIs

The APIs may be useful in the implementation of the required functionality.

- `rwlock(9)` - interface to read/write locks
- `copy(9)` - kernel copy functions
- `malloc(9)` - kernel memory allocator
- `pool(9)` - resource-pool manager
- `atomic_inc_int(9)` - atomic increment operations
- `atomic_dec_int(9)` - atomic decrement operations
- `atomic_cas_uint(9)` - atomic compare-and-swap operations
- `sysctl(3)` - get or set system information
- `queue(3)` - implementations of singly-linked lists, doubly-linked lists, simple queues, and tail queues
- `tree(3)` - implementations of splay and red-black trees
- `execvp(3)` - execute a file

2.8 Constraints

2.8.1 Uniprocessor kernels

The kernel zones implementation is not expected to be multi-processor safe and will only be tested on a uni-processor system running a GENERIC (not GENERIC.MP) kernel.

2.8.2 Process visibility

Only the sysctl interfaces used by ps, pgrep, and kill will be tested. Other mechanisms for interacting with processes such as ptrace, ktrace, systrace, process groups, sessions, libkvm using /dev/mem or /dev/kmem will not be tested and do not need to implement isolation between processes in zones.

3 Submission

Submission must be made electronically by pushing changes to a git repository on `source.eait.uq.edu.au`. In order to mark your assignment the markers will clone the “master” branch from your repository. Code checked in to any other branch of your repository will not be marked.

Your repository is named “comp3301-s1234567-a2” or “comp7308-s1234567-a2” depending on which course you are enrolled in. s1234567 is replaced with your UQ username.

The repository has been pre-populated with a checkout of the OPENBSD_5_7 tag from the OpenBSD CVS repository.

The repository may be cloned via the following command:

```
git clone s1234567@source.eait.uq.edu.au:comp3301-s1234567-a2 src
```

As per the `source.eait.uq.edu.au` usage guidelines, you should only commit source code and Makefiles.

The due date for this assignment is 8pm on Friday the 18th of September, 2015. Note that no submissions can be made more than 120 hours past the deadline under any circumstances. The time at which changes are pushed to the repository is considered the time of submission, regardless of when the commits within the repository were made.

4 Marking Scheme

WARNING If your code doesn't compile and run, you won't get credit for it.

1. For this assignment, break the functionality into stages or pieces that you can work on independently.
2. Get one piece fully functional and thoroughly tested.
3. Go on to the next piece.
4. If you've tried to implement everything but haven't completed any pieces, you won't get credit for anything.

Individual functionality will be checked, tested, and marked.

For each piece of functionality, percentages of marks will be allocated as:

- 0 - 20% Function not implemented, or no working code
- 20 - 40% Some simple but incorrect functionality
- 40 - 60% Significant problems, such as frequent crashes or infinite loops
- 60 - 80% Moderate problems, fails relatively often
- 80 - 100% Few or no problems

Total marks out of 100:

- Coding style, readability, organization [10]
- kernel configures and builds successfully [5]
- `make includes` and a `libc` build and installs [5]
- `src/bin/ps`, `src/usr.bin/pgrep`, and `src/usr.sbin/zone` build and install [10]
- kernel boots and runs [10]
- `zone create ZONENAME` works [10]
- `zone list` works [5]
- `zone exec ZONENAME ksh` works [10]
- forked processes inherit the zone from their parent [10]
- signals are restricted appropriately [15]
- `zone destroy ZONENAME` works [10]

5 Revisions

Changes to the assignment specification can be reviewed at https://source.eait.uq.edu.au/viewvc/comp3301-pracs/2015/assignment2.md?view=log.