Logical Resource Isolation in the NetBSD Kernel

Kristaps Džonsons
Swedish Royal Institute of Technology, Centre for Parallel Computing
kristaps@kth.se

Abstract

Resource isolation is a strategy of multiplicity, the state of many isolated contexts within a parent context. Isolated resource contexts have functionally non-isomorphic resource mappings: contexts with equivalent domain identities map to non-intersecting ranges in the resource co-domain. Thus, in practise, if processes a and b return different values to an equivalent identity (say, for the hostname), then the callee context, for this identity, demonstrates resource non-isomorphism. Although isolation is by no means a new study in operating systems, the BSD family offers few implementations, at this time limited to FreeBSD's Jail and, potentially, the `kauth(9)` subsystem in NetBSD. These systems provide a framework with which one may construct isolated environments by cross-checking and switching over credentials at the kernel's boundary. In this document, we consider a radically different approach to resource isolation: instead of isolating at the kernel boundary, we consider a strategy of collecting entire kernel sub-systems into contexts, effecting bottom-up resource isolation. This document describes a *work-in-progress*, although a considerable implementation exists[1].

1 Introduction

Resource isolation may strictly be defined as a non-isomorphic mapping between unique resource identities (the domain) and mapped entities (co-domain): multiple contexts, with the same domain identity, mapping to non-conflicting range entities. Instead of a single, global task context, where all tasks have a common mapping, resource isolation implies a set of contexts, with set members defined by the commonality of their mapping function. Our use of the term resource, in this regard, refers to *mutable* entities.

In practise, resource isolation provides Unix processes ("tasks") a different view of their environment depending upon the context. For example, a particular process a, if isolated, may only view processes in its calling context A, while another process, b, can only see processes in its context B. The contexts A and B are non-overlapping; in other words, no process in A can see into B and vice-versa. Conventional Unix environments, contrarily, are non-isolated (or isolated to a single context).

In this document, we use set notation to describe tasks and resources. We define a mapping function f to accept a resource identity x and produce an entity y. The set of resources available to a task is $f(x_0) \ldots f(x_n)$, or, equivalently, $y_0 \ldots y_n$ mapped from $x_0 \ldots x_n$. Resource isolation implies a set $F = \{f_0 \ldots f_k\}$, where the ranges of any two f_i are non-overlapping. Thus, in a traditional Unix system with a single, global resource context, $F = \{f_0\}$. This single context f_0 has an equivalent range and co-domain.

We consider each f to be a black-box within the kernel, and F defines the kernel proper. In practise, this decomposes into each $f \in F$ having a unique identifying credential; two resource requests, for example, for the hostname, correspond to a single f_i in the case of a single-context environment, and f_i and f_j in a multiplicity environment. We consider "resources" in the broad sense as system calls or in the fine sense as system calls combined with particular arguments.

The complexity of our example isolation scenario

[1]See http://mult.bsd.lv

is considerable: there are many elements entangled in a process. In order to be isolated, the set of contexts must be injective; in other words, a co-domain entity may be mapped from only one domain identity. A process abstracts considerable complexity: a process is composed of memory maps, file descriptors, resource limits, and so on. To isolate one process from another, all non-injective conditions must be removed. For example, although process a may not be able to signal process b, it may try to affect it by changing resource limits, or manipulating the controlling user. If any of these resources conflict, then isolation is not maintained. We call these conditions resource conflicts.

A resource conflict may be effected both directly and indirectly. In the former case, a breach could occur if a were able to signal b using a non-standard signalling interface (through, e.g., emulated system calls that bypass the isolation mechanism). In the latter case, improperly isolated user checks could allow a to affect b by changing user resource limits. Since $f(x_j) = y_c$ and $f\prime(x_j) = y_c$, the system is no longer isolated.

In this document, we'll propose a system that implements resource isolation in order to provide efficient multiplicity without sacrificing elegance. Before discussing the methods of this system, we'll consider the reason for its implementation; in other words, why we chose to implement a new design instead of re-using existing systems. In order to properly discuss these systems, we'll introduce an informal taxonomy for multiplicity systems.

2 Terminology

In this section, we refine our introductory notation. Since multiplicity has many different forms, it's difficult to introduce a generalised notation that covers all scenarios. In the previous section, we used the term context to describe the mapping $f \in F$. We now introduce the term "operating instance".

Definition If x is a resource identity (e.g., system call), and y_0 is the entity mapped by f_0, with f_1 producing y_1 where $f \in F$ and $y \in Y$, then we define each $f \in F$ as an operating instance in a multiplicity system if and only if there are no two $f_i \in F$ returning the same y for a identity x.

In practise, multiple operating instances in a multiplicity system may only exist if there are no conflict points where $f_i(x) = f_j(x) = y$. In the introductory section, we used the term resource isolation to define this property; in this section, we introduce the notion of operating instances as those entities with isolated resources.

The existence of an operating instance doesn't necessarily imply instance multiplicity: in a standard Unix system, there always exists one operating instance. The property of multiplicity arises when the co-domain Y is completely partitioned into ranges of $f \in F$, where no $y \in Y$ conflict.

Definition A system may claim *operating instance multiplicity* when there are multiple non-overlapping ranges in the resource co-domain, mapped from different $f \in F$.

In this document, we consider only operating instance multiplicity. There are other targets of multiplicity, like operating system multiplicity, which will be addressed only in passing.

3 Scenario

There are many scenarios involving multiplicity: service consolidation, testing, containment, redundancy, and so forth. We choose three common multiplicity scenarios that, as we'll see, span the diapason of multiplicity strategies. It's important to stress that this analysis is of operating instance multiplicity: we disregard, for the moment, scenarios calling for whole operating system multiplicity. This isn't the focus of this paper, as many well-known operating system multiplicity systems exist for the BSD family.

We'll primarily focus on a cluster computing environment. In this environment, we must allow for many distinct process trees with isolated resource contexts, preferably rooted at `init(8)`, all of which are completely contained from one another. There must be negligible over-head in the isolation mechanism; more importantly, these systems must start and stop extremely quickly, so as to offer fine-grained scheduling of virtual environments on the processor. There must be thousands of potential instances, all, possibly, running simultaneously on one processor. In an alternate scenario, some instances may have priority over others; some, even further, may be entirely suspended, then restarted later. This document focusses largely on the in-kernel isolation strategy for such a system.

We also consider a virtual hosting environment. Like in cluster computing, we account for the possibility of many instances competing for resources. Unlike in cluster computing, we consider that the run-time profile for these systems is roughly similar; thus, sharing of code-pages is essential. Further, while cluster computing stresses the speedy starting and stopping of instances, virtual hosting emphasis fine-grained control of resources which are likely to remain constantly operational to a greater or lesser extent.

In both of these scenarios, each process must have a conventional Unix environment at its disposal. We decompose the term “resource” into soft and hard resources: a *soft* resource is serviced by the kernel's top-half (processes, memory, resource limits, etc.), while a *hard* resource is serviced by a physical device, like a disc or network card. Our scenario doesn't explicitly call for isolation between hard resources (we consider this a possibility for future work); an administrator may decide which devices to expose by carefully constructing `dev` nodes.

4 Related Work

At this time, of the BSD operating system family, only FreeBSD Jail[3] offers a complete isolation mechanism. Jail attaches structures to user credentials that define guest contexts within a host context. Guests have a resource range that is a strict subset of the host; all guests have non-intersecting ranges while the host's range is equivalent to the co-domain (thus, is allowed to conflict with any guest). FreeBSD Jail structures isolate primarily in terms of soft resources: the only isolated hard resources are the network, console, and pseudo-terminal interfaces. The Jail system first appeared in FreeBSD 4.0.

NetBSD 4.0 includes `kauth(9)`[1], which orchestrates the `secmodel(9)` security framework. This system allows kernel *scopes* (resource identity categories) and their associated actions (resource requests) to be examined by a pool of *listeners*. Scopes and actions are correlated with calling credentials and relevant actions are produced. This doesn't provide isolation *per se*, but we can, assuming other changes to the infrastructure and an implementing kernel module, envision a sytem interfacing with `kauth(9)` to provide in-kernel isolation.

We specifically disregard NetBSD-Xen (and other full-system virtualisers, including the nascent DragonFlyBSD `vkernel(7)`) from our study, as the memory overhead of maintaining multiple guest images is considerable and violates our stipulation for many concurrent contexts. Both of these systems full into the category of operating system multiplicity systems: instead of resource isolation, these virtualise the hardware context invoked by the generalised resources of an operating system. The overhead of this virtualisation is considerable. In general, we disregard operating system multiplicity systems (such as QEMU, Xen, and so forth) due to the high practical overhead of hosting virtualised images or manipulating them.

Furthermore, we also discard the `ptrace(2)` and `systrace(2)` mechanisms and inheriting isolation systems. The latter is being deprecated from the BSD family, while the former (which may suffer from the same error as the latter) requires considerable execution overhead to orchestrate. The `kauth(8)` mechanism, which will be discussed, may be considered an in-kernel generalisation of these systems.

Lastly, this document does not consider non-BSD kernel isolation mechanisms, of which there are many. Most of these systems are implemented using strategies equivalent to FreeBSD Jail, enacting functional cross-checks, or as `kauth(9)` through security policies. Linux has several implemented systems, such as OpenVZ and VServer, and Solaris has Zones.

5 Issues

There are a number of issues with the available systems. The most significant issue with both available systems is the strategy of isolation: checkpoints within the flow of execution, instead of logically isolating resources in the kernel itself. The former strategy we call *functional* resource isolation, which is an isolation during the execution of conflict points.

In FreeBSD Jail, prisons are generally enforced by cross-checks at the system call boundary. For instance, a request to `kill(2)` from process $a \in A$ to $b \in B$ (as in our above scenario) is intercepted and eventually routed to `prison_check(9)` or similar function, which checks if the prison contexts are appropriate. In the Jail system, a host may affect guests, but guests may not affect each other or the host. Each potential conflict between process resources must be carefully isolated:

```
int
prison_check(struct ucred *c1, struct ucred *c2)
{
    if (jailed(c1)) {
        if (!jailed(c2))
            return (ESRCH);
        if (c2->cr_prison != c1->cr_prison)
            return (ESRCH);
    }
    return (0);
}
```

This function, or similar routine, must wrap each conflict point in order to protect the isolation invariant of the instances. In order for this methodology to work, each conflict point must be identified and neutralised. Clearly, as the kernel changes and new conflict points are added, each must be individually addressed.

The `kauth(9)` system enforces a similar logic. When kernel execution reaches a fixed arbitration point, zero or more listeners are notified with a variable-sized tuple minimally containing the the caller's credential and the requested operation (additional scope-specific data may also be passed to the listener). Thus, a signal from $a \in A$ to $b \in B$ may be intercepted with operation `KAUTH_PROCESS_CANSIGNAL`, and the listener may appropriately allow or deny based on the involved credentials.

The Jail system, regarding resource isolation, has a considerable edge over `kauth(9)`: the `kauth(9)` framework does not provide any sort of internal identification of contexts. A practical example follows: if one wishes to provide multiple Unix environments, there's no way to differentiate between multiple "root" users. `kauth(9)` listeners receive only notice of user and group credentials, and has no logic to pool credentials into a authentication group. This considerably limits the effectiveness of the subsystem; however, it's not an insurmountable challenge to modify the implementation to recognise context, although instance process trees would not be rootable under `init(8)`.

Although FreeBSD Jail does have an in-kernel partition of resources, the implementation falls short of full partitioning. Each credential, if jailed, is associated with a `struct prison` with a unique identifier. When a request arrives to the kernel on behalf of an imprisoned process, the task credentials have the standard Unix credentials and also an associated prison identifier. This allows conflicting user identifiers to be collected into prisons.

Our issue with both systems is the strategy by which isolation is enforced: functional isolation, where every point of conflict must be individually resolved. This is flawed security model, where in order to guarantee isolation, one must demonstrate the integrity of every single conflict point. Since some conflict points are indirect, this is often very tricky, or outright impossible. A cursory investigation of the `sysctl(3)` interface reveals that the `hostid` of the host system may be set by guests. Although this may be deliberate, the onus is on the developer to ensure that all conflicts are resolved; or if they are not, to provide an explanation.

6 Proposal

We consider an alternative approach to isolation that provides *a priori* isolation of kernel resources: logical isolation. Instead, for example, of cross-checking the credentials of process $a \in A$ signalling $b \in B$, we guarantee $A \cap B = \emptyset$ by collecting resource pools into the resource context structures themselves. In other words, the system resources themselves are collected within contexts, instead of requiring functional arbitration. Although this strategy is considerably more complicated to initially develop, the onus of meticulous entry-point checking is lifted from the kernel.

First, our method calls for a means of context identification. Like with Jail, we associate each credential with a context; in this document, we refer to this structure as the *instance* structure. Instance structures have one allocation per context and are maintained by reference counters.

In order to isolate at the broadest level, we comb through the kernel to find global structures. These we must either keep global or collect into the instance framework. Some resources, like the hostname and domainname, are trivial to collect. Others, like process tables, are considerably more difficult. Still others, like network stack entities (routing tables, etc.) are even more difficult. However, once these have been appropriately collected, we're guaranteed isolation without requiring complex border checks.

Further, we propose a forest of instance trees: instances may either be rooted at `init(8)` to create simultaneous, isolated system instances, or instead branch from existing instances, effectively creating a host/guest scenario. Child instances introduce some complexity; however, instead of building a selective non-injection into our original isolation model (as in FreeBSD Jail), we manage child instances through a management interface, instead of the violating our logical model. In practical terms, instead of issuing `kill(1)` to a child instance's process, a parent must operate through a management tool (described in "Implementation") with default authority over child operation.

Since the topic of hard resources (devices) is orthogonal to isolation as per our described scenario, we relegate this topic to the "Future Work" section of this document. The same applies for the proposed management interface.

7 Implementation

We focus our implementation on NetBSD 3.1. Our choice for this basis system was one of cleanliness, simplicity, and speed. FreeBSD proved to be too complex and already encumbered by the existing prison mechanism. OpenBSD, while simple and very well documented, can't compete with NetBSD (or FreeBSD) in terms of speed. NetBSD has proved to have very well-documented with concise code. The speed of the system is acceptable and the number of available drivers is adequate. Our choice of basis kernel is still open to change; the alterations, although extensive, are relatively portable among similar systems. From NetBSD we inherit a considerable set of supported architectures. Since this proposal doesn't affect the kernel's bottom-half, our project inherits this functionality.

The existing implementation, which is freely downloadable and inherits the license of its parent, NetBSD, carries the unofficial name "mult". The remainder of this document focusses on the design, implementation, and future work of the "mult" system.

7.1 Structure

The system is currently implemented by collecting resources into `struct inst` structures, which represent instances (similar to `struct prison` in FreeBSD's jail). Each instance has a single `struct inst` object. There are two member classifications to the instance structure: public and private. Public members are scoped to the instance structure itself; private members are anonymous pointer templates scoped to the implementing source file. What follows an abbreviated view of this top-most structure:

```
struct inst {
    uint               i_uuid;
    uint               i_refcnt;
    struct simplelock  i_lock;
    int                i_state;
    LIST_ENTRY(inst)   i_list;

    char               i_host[MAXHOSTNAMELEN];
    char               i_domain[MAXHOSTNAMELEN];

    inst_acct_t        i_acct;
    inst_proc_t        i_proc;
    ...
};
```

In this listing, `i_host` and `i_domain` are public members: their contents may be manipulated at any scope. The `i_acct` and `i_proc` members are private; their types are defined as follows:

```
typedef struct        inst_acct *inst_acct_t;
typedef struct        inst_proc *inst_proc_t;
```

The definitions for `inst_acct` and `inst_proc` are locally scoped to source files `kern_acct.c` and `kern_proc.c`, respectively. The `inst_proc` structure is locally scoped with the following members:

```
struct inst_proc {
    uint             pid_alloc_lim;
    uint             pid_alloc_cnt;
    struct pid_table *pid_table;
    ...
};
```

This structure consists of elements once with global static scope to the respective source file. The following is an excerpt from the pre-appropriated members in the stock NetBSD 3.1 `kern_proc.c` source file:

```
static struct pid_table *pid_table;
static uint pid_tbl_mask = INITIAL_PID_TABLE_SIZE - 1;
static uint pid_alloc_lim;
static uint pid_alloc_cnt;
```

Other private members share similar structure. Deciding which non-`static` members to make private, versus instance-public, is largely one of impact on dependent callers throughout the kernel.

At this time, there are a considerable number of appropriated subsystems with one or both public and private members: processes, accounting, pipes, kevents, ktraces, System V interprocess communication, exit/exec hooks, and several other minor systems. The procfs pseudo-file-system has been fully appropriated with significant work on ptyfs as well. Subsystems are generally brought into instances on-demand, that is, when larger, more significant systems must be appropriated.

7.2 Globals

There also exists a global instance context for routines called from outside a specific calling scope, for example, scheduling routines called by the system clock. These may need to iterate over all instances. The global list of instances may be accessed from a single list structure `allinst`, much like the previous `allproc` for process scheduling. Instances may also be queried by identifier, which is assumed to be unique. This convention is under reconsideration for the sake of scalability and the possibility of conflicting identifiers with high-speed cycling through the namespace of available identifiers.

7.3 Locking

Locking is done differently for different services. Private members often have their own native locking scheme inherited from the original implementation. Some public members also inherit the original locking, most notably the process subsystem, which has several public members as well as a private definition. General locking to instance members, those without a subsystem-defined locking mechanism, occurs with the `i_lock` member of `struct inst`. The global instance list has it's own lock, appropriate for an interrupt context.

7.4 Life-cycle

Instances are created in a special version of `fork1(9)` called `forkinst1(9)`, which spawns an instance's basis process from the memory of `proc0`. At this time, the instance structure itself is created. The private members are passed to an appropriate allocation routine defined for each member; usually, the memory referenced by these pointers is dynamically allocated.

The life-time of an instance is defined by its reference count, `i_refcnt` in `struct inst`. When this value reaches zero, the instance is cleaned up, with each private member being passed to a corresponding release routing, and returned to the instance memory pool. These allocation and deallocation routines are similar for each instance:

```
int
inst_proc_alloc(inst_proc_t *, int);

void
inst_proc_free(struct inst *);
```

The release of an instance's final process (usually the `instinit(8)` or `init(8)` process) must be specially handled. In normal systems, processes must have their resources reclaimed by the parent or init(8) under the assumption that init(8) never exits (or the system usually panics). This case is no longer true in our system, thus, a special kernel thread, instdaemon(9), frees the resources of these special processes. Since kernel threads traditionally reparent under the init(8) process, and this process is no longer singular, kernel threads also are reclaimed by instdaemoe(9).

7.5 Administration

There are several tools available with which one may interact and administer the instance framework. These interact with the kernel through either the sysctl(3) interface or a new system call, instctl(2). The former is primarily to access information on the instance framework, while the latter manipulates it. The instctl(2) function operates on a single argument, which contains parameters for controlling the instance infrastructure:

```
int
instctl(const struct instctl *);
```

The struct instctl structure allows several modes of control: debugging information, starting instances, and stopping instances. We anticipate this structure to grow significantly to account for other means of control.

There are several implementations of these functions. The instinfo(8) utility lists information about instances and the instance framework, instps(1) is an instance-aware version of ps(1), and instctl(8) which directly interacts with the instctl(2) system call.

An alternate init(8) implementation, instinit(8), is currently used for non-default instances. This is a temporary measure while the semantics behind terminal sharing are formalised. The instinit(8) process acts like init(8) except that it doesn't start the multi-user virtual terminal system. It's a required process for all starting instances, i.e., it must exist within the root file-system in order for the instance to "boot".

The system for administering instances is still under consideration, and may change. At this time, we chose simplicity, in this regard, over elegance and scalability; our focus is on the system's stability and design continuity. Administration is a topic that we wish to re-consider when a significant degree of scalability has been achieved, and the administration of thousands of instances becomes necessary.

8 Future Work

The "mult" system has a fairly well-defined short-term future, with some interesting possibilities for long-term development.

In the short-term, we anticipate fully appropriating pseudo-devices into instances. Furthermore, we envision a system for delegating physical devices to instances in a clean, elegant manner. Although not strictly-speaking a short- or mid-term goal, optional mid-term work, once the framework interfaces has been finalised, is to optimise the boot and shutdown sequence of instances. Lastly, inter-instance communication and manipulation should also be added to the control framework divvying physical resources between instances.

Pseudo-devices require fairly significant consideration. The most important is the network, followed closely by the terminal. We intend on following a similar path as the cloneable network stack for FreeBSD[4], where the entire network stack infrastructure is fully brought into instances. Bridges will allow instances to claim an entire network device, whose data may be routed to another instance controlling the physical network interface.

Terminals may be appropriated by dividing between terminal instances and non-terminal instances, where the former is connected to a real terminal and the latter runs "head-less". At this point, all instances run head-less, i.e., they have no controlling terminal during boot. This necessitated re-writing init(8) as instinit(8) to skip multi-user virtual terminal configuration. Each terminal instance would connect to one or more virtual terminals. Obviously, since virtual terminals are a scarce resource, this will rarely be the case; however, connecting instances to terminals allows multiple sets of monitor, keyboard and mouse connecting to the same computer and interacting with different instances.

There are a significant number of potential optimisations to the instance infrastructure. It's absolutely necessary that the scheduler and memory manager account for instance limits, allowing administrators to make real-time adjustments to the scheduling priority of individual instances. The Linux VServer uses a two-tiered scheduler for its instance implementation (which uses functional isolation): we envision a similar scenario but with an emphasis on real-time scheduling. Furthermore, we plan on introducing a per-subsystem configuration structure. At this time, each subsystem (processes, pipes, and so forth) is configured with the system defaults. By allowing these defaults to be changed, or disabled entirely, instances may be customised to their run-time environments.

With a completed isolation environment, we can begin to consider extending our model to support hardware. At this time, our model depends on an administrator to appropriately partition resources for each instance, much like with FreeBSD Jail. We envision extending isolation to the device level; instead of changing device drivers themselves, we consider an additional layer of logic at shared areas of abstraction code. Our approach is divided into two phases. First, we must have a means of reliably identifying devices; second, we must have a means to intercept data flow to and from these devices. Lastly, we must correlate device identity, operation, and credentials to arbitrate access.

We plan on drawing from the Linux kernel's `udev`[2] strategy to map devices to identifiers. Since the administrator will be creating the access module rules, there must be a means to flexibly label devices as they're recognised by the system. This strategy will involve our considerations of device pseudo-file-systems, which must be coherent with the notion of instances. This will govern the exposure of devices to instances, and considerably narrow the window of exposure.

Second, we must define arbitration points. These will most likely occur when requests are brokered to individual drivers. We plan on drawing on scope parameters from `kauth(9)` to find a generalised granularity. Realistically, most device isolation may be solved by an intelligent `dev` file-system. Both of these concepts, the device file-system and arbitration, must work together with the notion of instances. We propose an access module that arbitrates requests for hard resources, and limited interaction between other instance contexts. Since hard resources may not be logically isolated as may soft resources (as there's no natural correlation between a hard resource and a particular instance), the onus falls on the administrator to properly map instances onto devices.

9 Conclusion

In this document, we introduced the theory of isolation and discussed various existing strategies. After considering the limitations of these strategies, we proposed a new one. Our new strategy is not without its caveats: since the necessary altercations span the kernel diapason, keeping these sources synchronised with the origin kernel is a difficult task. However, since our changes primarily affect the kernel top-half, we believe that our sacrifice is warranted; we can still inherit additional drivers and sub-system changes from the main-line.

10 Acknowledgements

We'd like to thank the University of Latvia's Institute of Mathematics and Computer Science, where the system began development. Special thanks to the BSD.lv Project for its initial funding of the work, and a special thanks to Maikls Deksters for his support and contributions. Lastly, thanks to the Swedish Royal Institute of Technology's Centre for Parallel Computing for their continued support of this project.

References

[1] EuroBSDCon. *NetBSD Security Enhancemenents*, 2006.

[2] Greg Kroah-Hartman. udev: A userspace implementation of devfs. In *Proceedings of the Linux Symposium*, July 2003.

[3] SANE 2nd International Conference. *Jail: Confining the Omnipotent Root*, 2000.

[4] Marco Zec. Implementing a clonable network stack in the freebsd kernel. In *USENIX Annual Technical Conference, FREENIX Track*, 2003.