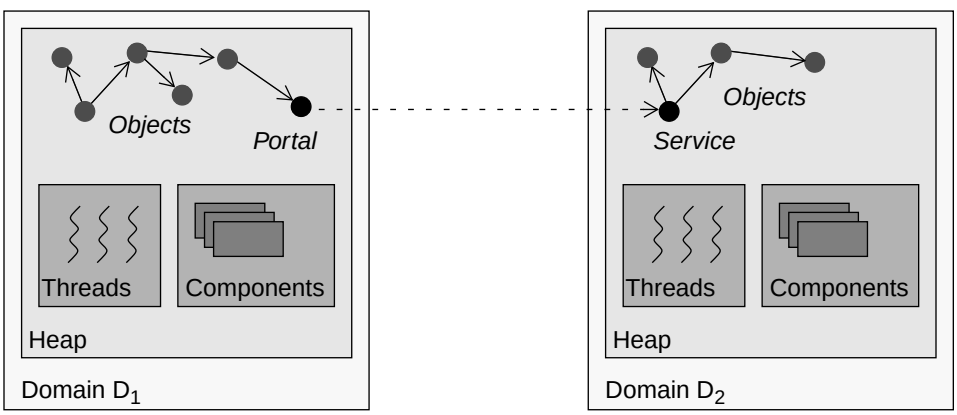
CHAPTER 4 *Domains*

This chapter describes the domain abstraction. It describes the lifecycle of a domain, its isolation properties, and the performance of domain related operations, such as domain creation.

1 Structure of a domain

Every domain contains a heap, a number of code components, threads, portals and services. The heap is used to store all data of the component, including thread controls blocks, stacks, portals, services, and components[1]. The heap memory is managed automatically by using a garbage collector.

Figure 4.1: Structure of a domain



2 Lifecycle of a domain

2.1 Creation

A domain is created by using a method of DomainZero's DomainManager service. Components are loaded into the domain or shared with other domains (see the Components chapter for details). After domain creation the domain is automatically activated by starting an initial thread. This thread starts

1. In the current prototype the components are stored in a separate memory area.

executing an entry method that is specified during domain creation. An entry method is a static method that must be part of the components that are loaded into the domain. This entry method can be passed an array of String parameters and an array of Object parameters. As portals are subtypes of Object they can be passed in the Object array. It is also possible to pass the naming service to the domain which allows the domain to obtain more portals. In a former version of JX this was the only way how a domain could obtain a reference to the naming service. This required that the naming portal be passed around in the system either by passing it to newly created objects as a parameter of the constructor or by storing it in a static variable. This turned out to lead to very cluttered programs and we introduced an easier way to obtain the naming portal. There is a static method InitialNaming.getInitialNaming() that is part of the zero interface component. This method has an empty implementation but the translator emits a call to a microkernel function whenever it translates an invocation of this static method. The microkernel function returns the naming portal of the domain that issued the call. The interface of the Naming portal is described in Figure 4.2.

Figure 4.2: Naming portal

The Naming portal has two methods: a method to register a portal and a method to lookup a portal. This represents the minimal functionality of a name service. Additional methods that allow, for example, to unregister portals or receive a notification when a new portal is registered, can be provided by name services that run outside DomainZero and use an extended Naming interface.

```
public interface Naming extends Portal {
    void registerPortal(Portal portal, String name);
    Portal lookup(String name);
}
```

2.2 Termination

A domain can be terminated *explicitly* at any time[1] using a kernel service. The security system decides whether one domain is allowed to terminate another domain. A domain terminates *automatically* when there are

- no running threads,
- no services,
- no references to threads from the outside[2], and
- no registered interrupt handlers.

When all these four conditions hold the domain has no internal activity and can not be activated from the outside.

When a domain is terminated, all threads are stopped, all services are deactivated, and active service invocations are terminated by throwing an exception to the calling domain. Threads that wait for the completion of a portal invocation are terminated by signalling the termination to the called

1. Device-driver domains can defer termination.
2. References to threads from outside the domain could be used to unblock a blocked thread. One use of these references is to unblock a thread by a timer service that runs in another domain. See Section 3.3 of Chapter 9 for details about the timer service.

domain, which then can decide to terminate the call at once or when it reaches a save point to do so. The reference counts of all memory objects (see Chapter 8, Section 3.1) and services (see Chapter 5, Section 5) that are referenced by portals of the domain are decremented. The fixed memory area and the heap are released.

It is difficult to safely terminate threads or protection domains in systems that support fine-grained sharing of objects. A thread can hold a lock for an unlimited amount of time. Terminating the thread while it holds the lock may leave the data structure in an inconsistent state. Rudy et al. [149] present a technique based on bytecode rewriting that can safely terminate a thread. These problems only appear because threads and data structures are not unambiguously associated with a protection domain. In JX objects as well as threads are cleanly separated into domains and domains share information by using portals. The use of a portal—the invocation of a method—is clearly visible in the source code. Every portal invocation can throw an exception, for example, when the server domain is terminated. Whether the data structures of the server domain become inconsistent is not relevant, because it is terminated anyway. The client domain can detect and react to a server termination by catching and handling the thrown exception.

3 Isolation

The domain is the unit of protection and resource management. It is isolated from other domains with respect to resource usage and data access:

- *Sandbox*: Every domain can only access its own data or methods or invoke methods that are exported by other domains.
- *Faults*: Due to a software bug or hardware error a domain can fault. The domain must be removed or restarted without affecting the rest of the system. All resources of the domain, such as interrupts and memory, must be freed.
- *Resource usage*: Every domain can use a granted resource without affecting other domains.

Fault isolation can be used to guarantee non-stop system execution. A (reliable) monitor module and a (reliable) specialized TCP implementation could be used for remote administration even in case of a complete fault of the remaining system.

3.1 Resource management

The unit of resource accounting is a domain. Once a resource, such as memory or CPU, is committed to a domain it can be used without further accounting. For example, a domain manages its heap with large memory blocks. Once such a block is committed to the domain it can use an arbitrary memory management scheme to allocate objects in this block.

Resource types. There are two fundamental resources: memory and CPU time. Because these two resources are necessary for every computation we call them *primary resources*. Depending on the hardware of the system there are additional resources, such as a disk, a network interface, a display, and input devices. We call them *secondary resources* because they require primary resources to be used. Built upon primary and secondary resources are *virtual resources*. They are a virtualization of

primary and secondary resources to allow (i) concurrent access by multiple users or programs and to allow (ii) easy and portable access to the resource by providing a high-level API.

The management of the two primary resources is explained in Chapters "Processor Scheduling" on page 73 and "Memory Management" on page 83.

When a domain obtains a portal to a service in another domain it acquires a virtual resource. When the domain terminates this resource must be released. This is done automatically when the reference count of the service drops to zero. The service may have allocated resources that are not released automatically. Therefore a method serviceFinalizer() is invoked at the service object when the reference count of a service drops to zero. This method is used for example in the window manager. The window manager creates windows for client domains. When such a client domain terminates the window service is deleted. All windows that belong to this client must be closed and associated resources must be released.

Shared services. When more than one domain obtains a portal to a service the service is shared between domains. Shared services usually implement high-level abstractions, such as files, or network connections. The implementation of these high-level abstractions maps the high-level virtual resource to lower-level virtual or physical resources. The domain that implements this mapping and provides the high-level resource is responsible for accounting the resource usage. How this accounting is done is outside the scope of the JX architecture. The resource container abstraction [13] can be used to separately account different services that run in the same domain but service different resource principals.

Protected data structures. Address-based systems must allocate data structures in the kernel address space[1], which requires a separate resource accounting mechanism for the memory that is consumed by these data structures. Type safety allows the kernel to maintain a data structure on the heap of a domain without the domain itself being able to access this data structure directly. We call these data structures protected data structures. Because protected data structures are stored on a domain's heap they do not consume any global resources.

3.2 Execution environment

A domain provides an execution environment for components. Components with differing requirements are run in different domains. An execution environment includes

Memory management. A domain can use a specific garbage collection strategy (or no garbage collection at all).

Scheduling and synchronization. A domain can use a specific thread management strategy, including a custom implementation of mutexes and condition variables, and a scheduling strategy (non-preemptive or preemptive, priority-based, deadline based).

Portal and service management. It can be specified per domain and per service implementation class how incoming service requests are handled, whether a new thread pool is created for a service, and whether a new service thread is added to the thread pool if a service thread should block.

1. An example are LPC message blocks in Windows 2000 [163].

Namespace. Code that runs in a domain needs portals to communicate with other domains. Initially a domain can access only a name service portal[1]. The name service for the initial domain, DomainInit, is provided by the microkernel, DomainZero. When a domain creates a new domain the creating domain can attach a new name service to the created domain, otherwise the name service of the creating domain is used. The current implementation of DomainInit creates a new name service and attaches it to all domains it creates. This guarantees that the name service of the microkernel is only used during bootstrapping the DomainInit naming service.

4 Domain ID

Several data structures in the microkernel and on a domains heap need to reference domains. Although it is sufficient to use a domain ID to unambiguously refer to a domain, using a direct pointer to the Domain Control Block (DCB) is much faster. In the current implementation DCBs are not moved but when a domain terminates the DCB of the domain is reused for domains that are newly created. This means that before a domain pointer can be used it must be checked whether it is still the DCB of the original domain. To be able to perform such a check the domain ID must be stored together with the domain pointer. Before the DCB is used this domain ID must be compared with the domain ID that is stored in the DCB. This requires the domain ID to be unique during the complete lifetime of the system, i.e. between two reboots. If a domain is created every nanosecond[2] a 64 bit range for domain IDs will last for 830 years. This means that the system must be booted at least once in 830 years.

5 Performance evaluation

Creation time. We measured the time to create a domain and compare it with the time to create a process in Linux. We created 100 domains and measured the time. On Linux we created 100 processes by using the fork system call and measured the time. We also used fork in combination with exec to load a new program. The results are presented in Table 4.1. JX domain creation includes the component loading and preparation (see Chapter 6 p. 59). It can be compared with a combination of fork and exec on UNIX. The factor of domain creation to Linux fork/exec is about 1.2 (Table 4.1).

Tab. 4.1 Domain creation time and destruction time

operation	time (µs)
JX domain creation	2245
Linux fork	263

1. As has been described in Section 2, during domain creation a domain is passed an object array that also may contain portals. This feature is a recent addition to the system and it is only necessary to reduce the startup time of domains and saving costly name service lookups. In principle, passing only the naming portal is sufficient.
2. Note that the domain creation time on our 500 MHZ PIII is about two milliseconds.

Tab. 4.1 Domain creation time and destruction time

operation	time (µs)
Linux fork/exec	1814

Memory footprint. Many applications, such as databases, agent platforms, and active networks must create large numbers of domains that each encapsulate a small amount of untrusted code. A small memory footprint of a protection domain is important for these applications. The memory overhead of a domain consists of the Domain Control Block, the domain specific code, the heap, and data structures that are allocated per-thread. Each thread needs a thread control block, which includes an area to save the CPU state and a stack. The stack consumes most memory and often the major part of this memory is unused. To avoid this situation a linked stack frame organization could be used. Stack frames are not taken from preallocated stack memory but allocated from the global memory pool and linked together. Although this approach does not waste memory for unused stack space it increases the method invocation overhead and can lead to fragmentation of the global memory system. Therefore we use a technique that is a combination of these two. We allocate a stack chunk that is relatively small. When there is no space to allocate a new stack frame we create a new stack chunk and link these two chunks together. When all stack frames of a stack chunk are freed, the chunk can also be released. To avoid allocation and deallocation of chunks that contain only one frame, deallocation must be delayed. It could be delayed until the previous chunk is half empty, but this would require a special check every time a stack frame is popped. Therefore we delay it, until the previous chunk is empty.

Figure 4.3 compares the memory requirements of a JVM process on Linux with the memory requirements of a JX domain. It is difficult to notice the JX line because a JX domain has a low memory footprint of 35 KB while a JVM process requires 3.4 MB. This data can be paged to secondary storage and only a part of it (the working set) needs to be resident in memory. But because of the large difference in DRAM and disk access speed paging severely affects performance.

The 35 KB can be split into 7 KB private code (including memory for static variables), 4 KB scratch memory used for portal parameter copying (see Chapter 5, Section 3.1), and 24 KB heap. The heap is populated by 10 KB character arrays that are created by the static initializers (for example by the Character class) and the “Hello World” application, less than 1 KB small objects, 8 KB for two stacks (4KB each for the initial thread and the garbage collector thread), 600 Bytes for two thread control blocks. This sums up to 20 KB. The remaining 4 KB are allocated but not used.These numbers indicate that scalability can still be improved by reducing the number of statically allocated character arrays and by reducing the stack size. One cost is not included in the 35 KB, that is the memory consumed by the Domain Control Block. The DCB contains storage space for the domain-local GC implementation (196 bytes), for the domain-local scheduler (128 bytes), and 484 bytes for domain management. Additionally it contains a list of services and a list of service thread pools[1]. These lists consume 4 * MAX_SERVICES + 4 * MAX_THREADPOOL bytes, where MAX_SERVICES is the maximal number of services exported by the domain and MAX_THREADPOOLS is the maximum number of thread pools for service threads (see Chapter 5 for details). Using a reasonable number of 50 services and 50 thread pools adds 400 bytes to the memory consumption of a domain.

1. In the current prototype these lists are statically allocated when a domain is created. They should also be placed on the heap.

Figure 4.3: Protection domain memory footprint

The UNIX/JVM graph shows the memory consumption (data segment) of UNIX JVM processes. The JX domains graph shows the memory consumption of JX domains that execute a “Hello World” program.

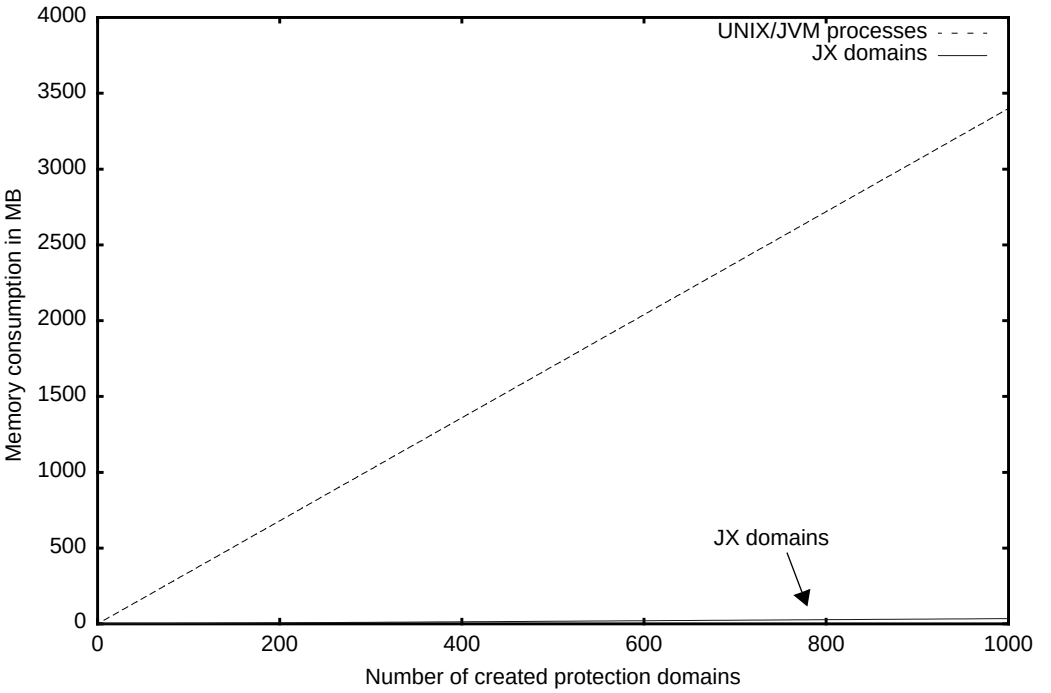
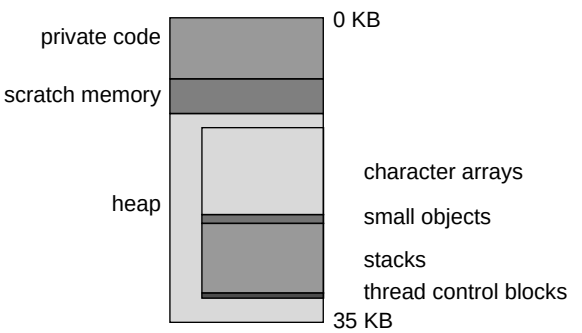


Figure 4.4: Break down of the memory footprint of a JX domain



6 Summary

This chapter described the domain abstraction. The domain is the unit of protection and resource management. A domain isolates code that is running in it against other domains. Each domain provides an execution environment that can be tailored to the needs of the code that runs in it. Domain creation is about 20% slower than fork and exec on Linux. The memory footprint of a domain is about one percent of the memory footprint of a JVM process on Linux.