Realizing Shared, Persistent Objects with Active Fields in C++

Krishna Bharat & Jeyakumar Muthukumarasamy

1. What are Shared, Persistent Objects with Active Fields ?

These are objects that are created in some address-space and may be accessed from other address-spaces thereafter. They are persistent in that they will continue to exist after the processes that created them have terminated. When the address-spaces are on different nodes, we refer to the shared objects as ‘distributed objects’. All processes (including the one that created the object) that use an object are said to be ‘importers’ of the object. There is no exporter of the object - since it is persistent and is always available on disk. If there is no shared file-system we may need an object-server which will act as the ‘exporter’ of the object.

The objects may have fields that are ‘active’. An active field is one which has a ‘notify procedure’ attached to it in some address-space where the object is imported. Notify procedures are procedures are invoked upon either a read or a write and either before or after the event occurred

1.1 Notification

There are 4 kinds of notifications that can be performed in response to accesses to active fields :

- *Notify Before Read:* The procedure is invoked and completes executing before the field is read. This is useful for lazy-updates to fields - they may be updated only when someone reads the field.
- *Notify After Read:* The procedure is invoked and completes executing after the field is read and before the next access to a shared object[1]. This is useful for book-keeping in a distributed setting, to keep track of who has read what.
- *Notify Before Write:* The procedure is invoked and completes executing before the field is written. This will allow locking and invalidation operations to be performed in preparation for the write, to maintain coherence.
- *Notify After Write:* The procedure is invoked and completes executing after the field is written and before the next access to a shared object[1]. This is useful for propagating changes to dependencies (e.g. constraint propagation) in response to the write.

[1] Or after a time-out period μ, whichever is earlier.

1.2 Naming & Universal Pointers

Objects are given names of the form <object-name>@<obj-repository-name>. If such an object does not exist it is created for the first time, or else the existing object is imported. In the current implementation object repositories are files and the object-repository-name is a filename.

Each shared object is thought to exist in a certain universal address range (universal because it is unique within a certain user-domain). There is a way (by a look-up) to map from a shared object name to a universal address range and vice-versa (i.e. from an address to the name of the shared object that contains it) unambiguously.

A universal address is 32 bit value with the most-significant-nibble being 0xF. It is assumed that these addresses are too large to represent a regular code or data segment address within a process and hence can always be recognized at run-time. The next 8 bits identify the object-repository (a file in this implementation) and the last 20 bits represent addresses within a virtual address-space implemented by a particular repository.

F	OR-Id	Offset in Vir. Addr Sp
4	8	20

Objects that are explicitly imported can have universal pointers to other shared objects and these get ‘implicitly’ imported as a result. Such objects are not considered imported until they are accessed for the first time. This avoids a cascade of imports.

2. C++ Semantics

In C++ objects are created as instances of some class. In the case of shared objects they need to be instances of a class derived from Shared_Object.

e.g. we could create shared object classes called A and B as follows :

```
class A : public Shared_Object {

....

int x;

};

class B : public A {

...
};
```

2.1 Instantiation

Shared Objects can only be created dynamically (i.e. using the `new` operator).

Instead of saying

```
A* a = new A(<constructor arguments>);
```

which is the normal C++ syntax for creating a new instance of class A called a, we would say

```
A* a = new(“foo@database1”) A(<constructor arguments>);
```

If there is no object called “foo” in “database1” it will get created. If “database1” does not exist it will be created as well. If either “database1” exists but cannot be read/modified (as revealed by the permissions), or it cannot be created a null pointer is returned.

The semantics of the constructor are changed slightly. The constructor gets executed each time the object is imported. If it is used to initialize fields the initialization will get repeated multiple times which may not be desirable. To distinguish between the a newly created object and an imported object we can make use of the fact that all bytes in data-fields in the object are initialized to 0 when it is created. So we could have an integer flag called ‘Initialized’ which is set to 1 by the first constructor to execute. This will provide the conventional at-most-once semantics of the constructor.

2.2 Attaching Notify Procedures

Notify procedures are of type `notify_proc` defined thus.

```
typedef void notify_proc(peer_descriptor p, event_type ev);
```

where `event_type` is one of `{BeforeRead, AfterRead, BeforeWrite, AfterWrite, BeforeAccess, AfterAccess}.`

Notify procedures are attached to instances typically in the constructor, as follows:

```
void attach_notify_proc(void* field, int size, event_type
ev, notify_proc np);
```

where `field` and `size` are typically the starting address and the size of a field, as in

```
attach_notify_proc(&(a->x), sizeof(a->x), BeforeRead, fun);
```

2.3 Buffered Access

Since there is a considerable overhead for accesses to shared objects we introduce the notion of a buffered pointer to shared objects.

The buffered pointer for any object can be obtained using the BUFFERED() macro.

```
e.g. A* a = new("foo@db1")A, b = BUFFERED(A);
```

In the above example `b` is a buffered pointer to the same object that `a` points to. However reads to `b` will not be trapped (hence they are termed quick-reads) and writes to `b` will not propagate to the version pointed to by `a` (the non-buffered pointer) and to copies in other address-spaces until the `_FLUSH()` method is invoked, as in

```
b->x = 1; b->y = b->x + 2; b->_FLUSH();
```

When the `_FLUSH()` method is invoked changes will be propagated to all copies, and notify procedures for writes will be executed - at most once per field. For completeness we have the complementary macro TRAPPED() which accesses the unbuffered pointer.

```
e.g. A* c = TRAPPED(b); /* c points to same version as a */
```

2.4 Tagging Pointer Fields

The shared object runtime ensures that writes to fields are propagated to other copies as-is. However if the field contains a pointer, equivalence rather than equality is the desired relation between values in different address-spaces. The programmer identifies such fields explicitly thus:

```
TAG_FIELD(&(a->y), SHARED_OBJ_PTR);
```

This happens typically in the constructor. Currently this is only applicable to shared object pointers. In the above example any value assigned to a->y is assumed to be a pointer to a shared-object in the local address-space and it gets mapped to the corresponding local pointers when the field is read in other address spaces. This is achieved by translating the local pointer at the time of assignment to a universal pointer which is mapped to a local pointer when it is read.

3. Implementation

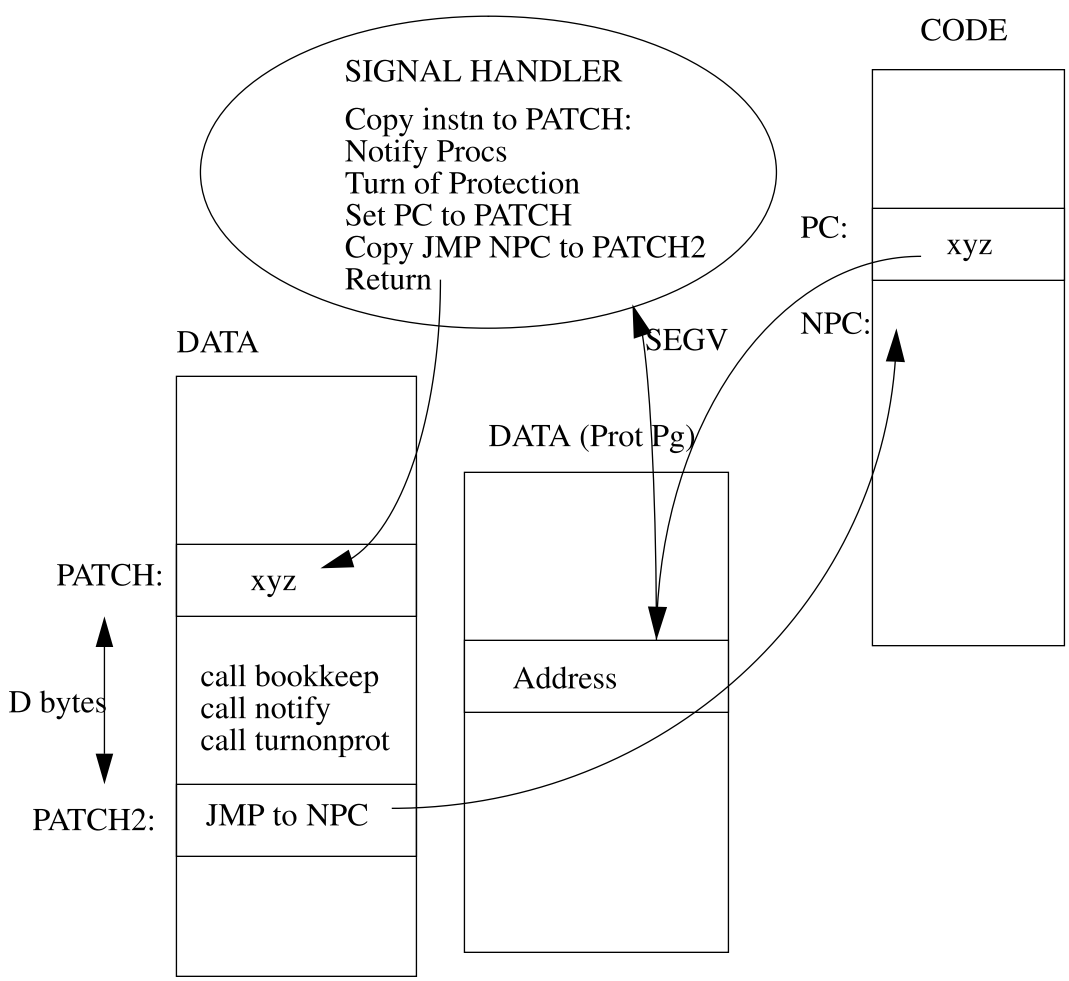
3.1 Trapping Accesses

Trapping accesses may be done at (i) Language level (ii) Preprocessor Level or (iii) OS level. In C++ there are numerous ways to lose type information (by casting or otherwise subverting the type system) and no scheme that works at levels (i) or (ii) can hope to be robust unless it is hopelessly inefficient (i.e. all accesses are prefixed by checks). At the OS level we can monitor accesses to `Shared_Objects` alone by creating them on a separate (protected) page in memory. This is the basis of our implementation.

We overload the `new` operator for `Shared_Objects` and allocate memory in a read and write protected page of memory. Currently all `Shared_Objects` are persistent so we `mmap` the file (or object repository) that contains (or will contain from now on) the object instance.

Accesses to the `Shared_Object` results in a segmentation fault which is handled by a signal handler. The signal handler first checks to see if the access occurs within a shared-object. Otherwise the default handler is called or the process crashes. The signal handler context provides the context of the segmentation fault - i.e. the `PC` value, the `NPC` value, the address where the error occurred and the registers.

We look at the opcode of the instruction at the PC value and determine if it is a read or a write. Correspondingly we perform notification if necessary. Then we relax the protection on the page, copy the bytes from `PC` to `NPC`-1 to a special part of data memory labeled `PATCH` and return from the signal handler with PC set to the address of `PATCH`. This causes the context to be restored and control to jump to the instruction at `PATCH` (which is the same instruction). Having executed the instruction the code placed after it calls a routine that does book-keeping and turns of protection. If there are notify procedures defined 'after' the event, we initiate a time-out (`SIGALRM`) at this point. Control returns to the original `NPC` value and the execution proceeds normally.



3.2 Implementing Buffered Pointers

The new operator when invoked allocates two pieces of memory - one in a protected page and one in a non-protected page (the buffered copy). The two copies are kept synchronized by the runtime whenever writes occur to the non-buffered page. However writes to the buffered page do not propagate to the non-buffered page instantly. They propagate only when the pointer if `FLUSHed` explicitly.

3.3 Handling Virtual Functions

C++ objects do not nest function pointers. The compiler uses type information to resolve the function address at compile time and passes the object to the function as an argument. In the case of virtual functions the function to be called is resolved at runtime. The pointer to the function is located by performing a lookup in a table called the virtual function table. The object contains a pointer to this table.

In the case of Shared_Objects the pointer to the virtual function table (which will be different in each address-space) is located at a fixed offset into the object namely `(sizeof(Shared_Object) - sizeof(void *))`. This is true for all derived classes as well. When the object is created the constructor initializes the virtual function table pointer in the protected page. This is trapped and copied to the buffered copy. The virtual function table pointer in the protected page is likely to change because the page is `mmaped`. Thereafter whenever there is a read to this field in the protected page the read is trapped and the correct value is copied over from the other copy before the read is allowed to complete.

3.4 Handling Universal Pointers

Universal pointers are pointers to other shared objects nested within the object. These need to be explicitly tagged as explained earlier. When a shared object is swapped in and used for the first time, the tagged pointers are resolved. If the corresponding objects exist they do not need to be swapped in. The corresponding pointer in the buffered copy is made to point to the appropriate object in the local address space. When reads occur in the protected page, if it is a read to a tagged pointer the real pointer is copied over from the buffered page, the read is allowed to complete and the original pointer is written back to the slot. If the object does not exist it is swapped in and marked as '`Unused`'. The first time it is accessed all the pointers within it are resolved in a similar manner and it is marked '`Used`'.

Assignments to a tagged pointer field always happen in terms of the local address space. The runtime traps these assignments and maps the local pointer value to a universal pointer value and stores it in the protected copy. This is explicitly propagated to other address-spaces which may need to swap the object in if not already present and update their corresponding entries in their buffered copies.