PatchWorks

High-Level Low-Level: Object-Oriented Patching.

by Patrick C. Beard.

In memory of Robert Herrell.

ABSTRACT

This document describes PatchWorks, an object-oriented trap patching system created by Robert (Mouse) Herrell and Patrick Beard. PatchWorks simplifies trap patching by handling many of the mundane chores that all trap patches must perform. The details of how patches are installed, and the assembly language “glue” required to implement patches, are taken care of, leaving the programmer free to decide what behavior a patch should perform. PatchWorks supports more than simple trap patching; it supports other patching mechanisms, such as interrupt and exception vector patching, and special purpose mechanisms for debugging and exceptional situations.

PatchWorks also speeds up the development cycle by providing a way for trap patches to be developed as applications, which removes the necessity of rebooting while developing patches. The process of developing operating system extensions is greatly enhanced with the use of PatchWorks. Now the benefits of object-oriented programming can now be used by systems level programmers.

While its implementation is specific to the Macintosh operating system, PatchWorks is written in portable C++ and could be ported easily to other platforms that provide some kind of patching mechanism.

INTRODUCTION TO PATCHWORKS

Background

The Macintosh operating system provides a mechanism by which the operating system itself can be replaced or modified. Apple uses this mechanism to fix bugs that are found in the ROMs in every Macintosh. Developers can use this mechanism to implement system enhancement utilities, often called “INITs” or “extensions.” The extension mechanism is documented in Inside Macintosh volumes IV, V, and VI, and the interested reader can look there for more details. The rest of this paper assumes an understanding of these chapters.

Motivation

PatchWorks was developed to ease the difficulty of writing operating system extensions. At the heart of all extensions are small pieces of code called *trap patches*. Patches are notoriously hard to write, debug, and

maintain. When Apple releases new system software, system extensions are usually the first software that breaks. PatchWorks reduces the process of writing a trap patch to the task of writing a C++ class that represents the behavior the developer wishes to instill in a particular system extension. Using a class to represent a patch has several advantages. Classes represent a combination of code and data, so a patch need not use global variables. Inheritance makes it possible for families of patches to share common behaviors. Since much code can be shared, compatibility problems are easier to solve.

Getting Started With PatchWorks

To show how easy it is to use PatchWorks to write a system extension, here is a simple example extension that comes with PatchWorks:

```
class MenuSelectPatch : public GenericPatch {
public:
      MenuSelectPatch();                              // constructor.
      virtual void Behavior();                  // called when patch is hit.

private:
      long itsCallCount;                              // keeps track of # of calls.
};

struct MenuSelectParameters {
      Point itsStartPt;
      long itsResult;
};

typedef pascal long (*MenuSelectProcPtr) (Point pt);
```

With the PatchWorks framework, a developer merely declares a sub-class of class `GenericPatch`, implements a constructor (and destructor if required), and the virtual function `Behavior`. To access the parameters passed to the trap, a structure is used to represent the parameters as they are pushed on the stack, including the pascal result if any. The contents of the stack frame are made available to the patch object through an inherited data member called "`itsFrame`."

The constructor is written like this:

```
MenuSelectPatch::MenuSelectPatch()
{
      // initialize instance variables.
      itsCallCount = 0;

      // install the appropriate patch.
      GenericPatch::InitGenericPatch(_MenuSelect,
            offsetof(MenuSelectParameters, itsResult));
      Install();
}
```

The call to `GenericPatch::InitGenericPatch()` prepares the patch to be installed. The first argument is the trap number of the trap to be patched. The second argument is the size of the parameter stack. This value is used

for tail patches to compute how many arguments to remove from the stack when returning to the caller. `Install()` actually installs the patch.

Next comes the virtual function `Behavior()`:

```
void MenuSelectPatch::Behavior()
{
      KeyMap keys;

      // up the call count.
      ++itsCallCount;

      // if control key held down, print a message.
      GetKeys(keys);
      if (keys[1] & 0x8) {
            MenuSelectParameters* params;

            // access the passed-in parameters.
            params = (MenuSelectParameters*)itsFrame->parameters;

            // print out where the mouse was clicked and how many
            // times we've been called.
            dprintf("MenuSelect:  startPt = (%d, %d), %ld calls.",
                        params->itsStartPt, itsCallCount);
      }
}
```

This function first increments a counter (behaviors typically do something more substantial like allocate memory, etc), and if the control key is held down, it accesses the trap's parameters and prints them. This is the "hello world" of trap patching examples.

UNDER THE HOOD

Figure 1 shows the hierarchy of classes that PatchWorks provides, plus where our example class fits into the framework.

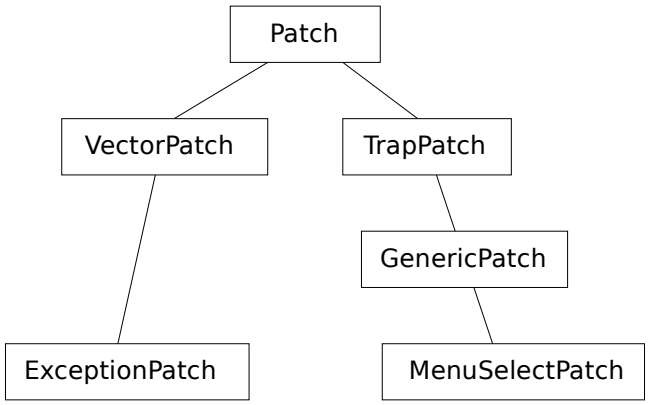

Figure 1. PatchWorks Class Hierarchy

The base class of all PatchWorks classes, `Patch`, is an abstract base that represents what is common to all patch objects. This abstraction provides a protocol for patching: all patches require glue (code that acts as an "agent" between the operating system and the patch object); a way to get the address of the routine that is being patched; and a way to set the new routine. Class `Patch` provides default glue, and defines the interface for getting and setting of patch addresses. The following is the declaration of class `Patch` from the file `Patch.h`:

```
class Patch {
protected:
      Patch();                                                          //
constructor protected to
                                                                              //
prevent direct use.
public:
      virtual ~Patch();                                          // destruction is
allowed.

      void Install();                                            // install the patch.
                                                                              //
delete to remove.
      static void RemoveAll();                                 // remove all installed
patches.

      void Enable();                                                  // enable the
patch.
      void Disable();                                            // disable the patch.
      PatchState Switch(PatchState state);      // get & set state.

      void* operator new(size_t n);                    // memory allocator.
      void operator delete(void* p);                   // memory deallocator.

protected:
      virtual PatchProcPtr GetAgent();                 // returns pointer to glue.
      virtual PatchProcPtr Get();                      // retrieve the old address.
      virtual void Set(PatchProcPtr proc);      // set the new address.

protected:
      static Patch* theirList;                                 // list of all installed
patches.
      Patch* itsNext;                                                 // next patch after this
one.

      PatchProcPtr itsBehavior;                                // routine to call for patch.
      PatchProcPtr itsOld;                                           // old routine to call.
      PatchStub* itsStub;                                            // the stub code.

      void* itsGlobals;                                              // pointer to globals.
      Boolean itsInstalled;                                    // if patch was ever installed.
      PatchState itsState;                                           // state of patch
(on/off)
};
```

Since class `Patch` is an abstract base class most of the patching work is performed by sub-classes of class `Patch`. Sub-classes provide patching "action" by overriding the virtual functions `Get()` and `Set()`. A patch is installed by a call to the `Install()` function, which

calls `Get()`, stores the result in the data member `itsOld`, and finally sets the new routine by calling `Set()`. The base class `Patch` provides this service of saving old patch addresses and installing new ones, while the details of how the patch routine addresses are retrieved and applied are provided by sub-classes. This allows other kinds of patching mechanisms to be installed without changing the basic protocol. Other mechanisms are discussed below.

Class `Patch` also provides functions to enable and disable your patch. This is accomplished by modifying the beginning of the patch glue code to jump directly to the old routine. On machines that have both instruction and data caches this requires a flush of both caches to keep things consistent. For this reason, it is unwise to enable and disable your patch in a tight loop, or any place where performance is critical. This capability is provided mainly to help you implement a control panel interface that disables your extension. (See the Appendix for more details.)

All patches that are installed with PatchWorks are kept in a list. This makes it possible to remove all of the patches that are installed with a single call, `RemoveAll()`. Once this call is made, it becomes possible to dispose all of the patch code an extension has loaded. This allows extensions to have a very small memory footprint. When patches are removed, they leave behind a 10-byte stub that links to the old trap address.

Sub-Classes

As figure 1 shows, several sub-classes of class `Patch` are provided. Class `TrapPatch` implements trap patching using the Macintosh toolbox, while `VectorPatch` implements patching of low-memory vectors, such as exception vectors. Class `GenericPatch`, a sub-class of `TrapPatch`, provides an object-oriented way of representing the behavior of a trap patch. The declaration of `GenericPatch` follows:

```
class GenericPatch : public TrapPatch {
public:
      void InitGenericPatch(short trap, long resultOffset);
protected:
      // override Behavior to do your stuff,
      virtual void Behavior();
      // or alternatively provide address of routine to call.
      virtual ProcPtr GetBehavior();
      // utility functions.
      void AbortTrap();             // call to return directly to the caller.
private:
      static void Dispatch(PatchFrame frame);
protected:
      PatchFrame* itsFrame;   // current stack frame.
      long itsResultOffset;   // offset to the result value on stack.
};
```

When a patch is derived from `GenericPatch`, the behavior of the patch is implemented by a call to the virtual function `Behavior()`. The protected data member `itsFrame` points to the stack frame the patch was called in. This stack frame contains all of the information that a patch is likely to need to perform its behavior. A `PatchFrame` is declared as follows:

```
struct PatchFrame {
      Patch* patch;                 // pointer to current patch object.
      void* offset;                 // magic address for tail patching.
      long rd0;                     // caller's d0-d2/a0-a1 which might contain
      long rd1;                     // parameters which patch can change.
      long rd2;
      void* ra0;
```

```
        void* ra1;
        void* ra4;                    // caller's a4 & a5 in case we use a4 globals.
        void* ra5;
        void* link;                   // link to previous stack frame. 28(sp)
        void* old;                    // old address that will be returned to.
        void* caller;                 // caller's address. 36(sp)
        char parameters[1];     // array of parameters (if any)
};
```

For register based traps, registers d0-d2, and a0-a1 are used to pass parameters and return results. The PatchFrame structure holds the saved values of the registers on patch entry, and the values they will be restored to on patch exit, so the patch can interpret these registers and change them as it sees fit. For stack-based traps, the array `parameters[]` represents the passed-in parameters. To access these parameters, the programmer creates a structure that represents the layout of the stack for that routine, and casts `parameters[]` to be a pointer to that structure. These are the actual parameters passed in to the trap by the caller, so if the trap changes them, they will be changed for the call to the actual trap. For example, a patch to `GetNextEvent` would use the following structure to represent the stack layout:

```
pascal Boolean GetNextEvent(short mask, EventRecord* event);

struct GNEParams {
      EventRecord* itsEvent;
      short itsMask;
      Boolean itsResult;
};
```

Notice that the data members of `GNEParams` are in reverse order from the parameters to `GetNextEvent`. This is because Pascal calling conventions dictate that parameters be pushed on the stack from left to right, with the result being a "hidden" argument that is pushed first. Since the stack grows down on the MC68000, the second parameter, `itsEvent`, is lower in memory than the first parameter, `itsMask`.

Head & Tail Patching

Finally, if the behavior routine wishes to head off the trap (i.e. to return to the caller directly) it should call the method `AbortTrap`(), which modifies the `PatchFrame` to return directly to the caller. This can also be used to implement tail patches. First the behavior routine calls the old routine (using the provided function `CallOS`() for register-based traps, or directly by typed procedure pointer) then calling `AbortTrap`() to return directly to the caller.

ADVANCED TOPICS

Global Patching

One of the techniques the author has developed when writing extensions deals with the problem of how to patch traps that the process managers of MultiFinder and System 7 patch destructively (that is they do not call patches that are installed at INIT time). One of these traps is `WaitNextEvent`, and the list seems to be growing. A technique that I have found effective is to defer patching until the Finder is launched and calls `InitGraf` (`LoadSeg` of CODE 1 in the Finder would work just as well). The essential trick is to cache away copies of the addresses of `SetTrapAddress` & `GetTrapAddress` as they are at INIT time and to use these directly rather than the ones installed by the system after INIT time. This provides post-INIT-time global patching, because one is essentially patching behind the system's back. The reason this works is that the Process Manager keeps track of what traps are patched in a given processes context by keeping track through a patch to `SetTrapAddress`. It builds a table of differences that must be applied to the trap dispatching table whenever a process context switch occurs. When INIT time `SetTrapAddress` is used, this process is defeated. (Caveat: it is anybody's guess as to how long this technique will continue to work. However; it has survived several system releases and to my knowledge is as good a method as any other I've seen. It's nasty, but it works. I have also spoken to an Apple engineer who could not find any fault with this technique.)

PatchWorks provides a class that implements this behavior called `GlobalPatch`. `GlobalPatch` is inherited from `GenericPatch`, and overrides the get/set routines provided by `TrapPatch`.

Extension Development With Scalper

Scalper is a debugging tool for writing extensions. Scalper patches *all* traps in the system, and allows an application to install patches globally (see discussion of global patches above), even under MultiFinder or System 7. An application registers with Scalper by calling a special Gestalt selector which returns a routine that installs patches into the linked list. A MacsBug dcmd called "unpatch" can communicate with Scalper to remove patches in the case of an emergency such as a crashed application. Using Scalper, patches can be debugged in a limited way using a source level debugger.

THE FUTURE OF PATCHWORKS

The current version of PatchWorks is compatible with THINK C 5.0 & MPW C++ 3.2. PatchWorks started as research project within Berkeley Systems, and I

will continue to improve and generalize it. However; no outside developer support is planned. The software is available to anyone that wants to use it. Developers are free to incorporate PatchWorks code into their products, providing they mention that PatchWorks was used, and to mention the authors.

Possible Improvements

The code that provides the basic trap patching services could ideally be made into some sort of shared library that all extensions could share. This would save memory and could open up some other interesting possibilities. One interesting possibility is the ability to develop system extensions that are installed and removed purely by double-clicking and quitting. In essence, extensions could become applications, and thus first class citizens of the Macintosh. PatchWorks combined with an enhanced version of Scalper, would provide a scheme by which patches can be applied and removed dynamically, without fear of the problems that current trap patching schemes have. The way this would work is that PatchWorks would maintain the calling links between all of the trap patches. Apple should consider creating a “Patching Manager” to the Macintosh operating system that provides these services.

APPENDIX - UNIVERSAL PATCH GLUE

The heart of the object-oriented patch mechanism is universal glue code that provides a uniform interface between the operating system and higher level patch code. This glue is used by every patch. Patches allocate a small “stub” that jumps to the shared glue. This stub has the following structure:

```
struct PatchStub {
      short itsJsrJmp;
      PatchProcPtr itsAgent;
      Patch* itsPatch;
};
```

When the patch is active, the variable `itsJsrJmp` is a JSR to absolute address instruction, and `itsAgent` is a pointer to the universal glue routine. The following code shows how PatchWorks’ patch glue is implemented. For every patch installed by PatchWorks, a copy of this code is created in the System heap. Two main services are provided by the glue: a pointer to the patch object is stored pc-relative in the glue; the globals register is automatically maintained for the patch code. Additionally, the glue supports tail patching by immediately returning to the caller.

```
;
;     PatchAgent
;
;     This routine is meant to be called from a small stub of code that has
;     this C structure:
;
;     struct PatchStub {
;           short itsJsrJmp;
;           PatchProcPtr itsAgent;
;           Patch* itsPatch;
;     };
;
;     This allows the patch agent code to pull out the instance by the
;     return address and reduce the per patch overhead.
;
PatchAgent  PROC  EXPORT
            WITH  Patch                                                ; use the Patch
object.
            link  FP, #0
            movem.l     d0-d2/a0-a1/a4-a5, -(sp)      ; save the registers.
            move.l      4(FP), a0                                ; return address points
into PatchStub
            move.l      (a0), a0                                       ; get pointer to
Patch out of it.
            move.l      itsOld(a0), 4(FP)                  ; put old trap address on
stack.
            move.l      itsBehavior(a0), d0                ; get the behavior.

            clr.l -(sp)                                                ; save room for
stack adjustment.
```

```
            move.l      a0, -(sp)                                   ; pass Patch instance.

            move.l      itsGlobals(a0), GLOBALS     ; set up the patch's globals.
            move.l      d0, a0
            jsr         (a0)                                        ; call the behavior
routine.
```

```
            add.l #4, sp                                        ; remove patch
parameter we pushed.
            move.l      (sp)+, d0                               ; see if we're returning
to caller.
            beq.s @restore

            ; We're heading off the patch, return to caller.
            ; This is accomplished by copying the return address back
            ; before where the parameters are, and adjusting the frame pointer
            ; to remove the caller's arguments if there are any.
            ;
            ; Note, the 36 & 28 are hard coded offsets to the link & return
            ; addresses. They are derived from the PatchFrame structure in
            ; Patch.h.
            ; In the case of exceptions, the caller is at 38 because of the
            ; status value. Therefore, we can't really tail patch exceptions
            ; with this glue.

            move.l      d0, FP
            move.l      36(sp), -(FP)                   ; get caller's return address.
            move.l      28(sp), -(FP)                   ; get old value of FP so unlk
works.

@restore
            movem.l     (sp)+, d0-d2/a0-a1/a4-a5
            unlk  FP
            rts

@macsbug
            dc.b  $80 + $A, 'PatchAgent'        ; macsbug symbol.

            ENDP
;
        END
```