Mac OS X Port Dtrace Probe Implementation

Dtrace was implemented under Solaris 8, which ironically did not support dtrace. Instead, a version of dtrace was hacked into the Sun's (now Oracle's) build machines. This dtrace version supported an early version of the USDT user space dtrace probes that I'll call USDT1.
When Apple introduced dtrace into its Leopard OS, it used a later version of Sun's dtrace that has a different probe style. Oracle has not been able to move to the newer probe style because it still has builds that require the older versions of Solaris.

The USDT1 probes require a bunch of macros in the code (src/share/vm/utilities/dtrace.hpp):

```
#define HS_DTRACE_PROBE_FN(provider,name)\
  __dtrace_##provider##___##name
#define HS_DTRACE_PROBE_DECL_N(provider,name,args) \
  DTRACE_ONLY(extern "C" void HS_DTRACE_PROBE_FN(provider,name) args)
#define HS_DTRACE_PROBE_DECL5(provider,name,t0,t1,t2,t3,t4)\
  HS_DTRACE_PROBE_DECL_N(provider,name,(\
    uintptr_t,uintptr_t,uintptr_t,uintptr_t,uintptr_t))
#define HS_DTRACE_PROBE5(provider,name,a0,a1,a2,a3,a4)\
  HS_DTRACE_PROBE_N(provider,name,((uintptr_t)a0,(uintptr_t)a1,(uintptr_t)a2,\
    (uintptr_t)a3,(uintptr_t)a4))
```

A typical USDT1 probe is shown below.

```
#ifdef USDT1
HS_DTRACE_PROBE_DECL5(hotspot, monitor__wait, jlong, uintptr_t, char*, int, long);
      HS_DTRACE_PROBE5(hotspot, monitor__wait, jtid, (monitor), bytes, len, (millis));
#endif /* USDT1 */
```

USDT2 probes are much simpler:

```
#ifdef USDT2
      HOTSPOT_MONITOR_WAIT(jtid, (uintptrt)(monitor), bytes, len, (millis));
#endif /* USDT2 */
```

We can dispense with all of the macro definitions and setup because the dtrace command (/usr/sbin/dtrace on Mac OS) takes care of the details. From hotspot.d, which defines the probes,
it translates the code:

```
provider hotspot {
  probe monitor__wait(uintptr_t, uintptr_t, char*, uintptr_t, uintptr_t);
```

and generates the following macro in hotspot.h (via dtrace.make, early in the hotspot build). Thus,
we end up with a simpler call and all the support macros are built automatically.

```
#define        HOTSPOT_MONITOR_WAIT(arg0, arg1, arg2, arg3, arg4) \
do { \
        __asm__ volatile(".reference " HOTSPOT_TYPEDEFS); \

__dtrace_probe$hotspot$monitor__wait$v1$756e7369676e6564206c6f6e67$756e7369676e6564206c6f6e67$63686172202a$756e7
369676e6564206c6f6e67$756e7369676e6564206c6f6e67(arg0, arg1, arg2, arg3, arg4); \
        __asm__ volatile(".reference " HOTSPOT_STABILITY); \
} while (0)
```

While inserting the USDT2 probes in the OpenJDK code base, I tried to make minimal disruptions to the USDT1 probes.
However, there are two complications.

First, many of the dtrace probes are wrapped in macros. The example above looks like this:

```
#ifdef USDT1
#Define DTRACE_MONITOR_WAIT_PROBE(monitor, klassOop, thread, millis)       \
  {                                                                        \
    if (DTraceMonitorProbes) {                                            \
      DTRACE_MONITOR_PROBE_COMMON(klassOop, thread);                       \
      HS_DTRACE_PROBE5(hotspot, monitor__wait, jtid,                       \
                       (monitor), bytes, len, (millis));                   \
    }                                                                      \
  }
#endif /* USDT1 */
```

Due to the complications of nested macros and my inability to do test builds on Solaris 8, I elected to duplicate the entire macro when translating to USDT2. This results in significant amounts of duplicated code. If the old style probes are never going to be updated, this code should probably be consolidated by restructuring the macros.

```
#ifdef USDT2
#define DTRACE_MONITOR_WAIT_PROBE(monitor, klassOop, thread, millis)       \
  {                                                                        \
    if (DTraceMonitorProbes) {                                            \
      DTRACE_MONITOR_PROBE_COMMON(klassOop, thread);                       \
      HOTSPOT_MONITOR_WAIT(jtid,                                           \
                           (uintptrt)(monitor), bytes, len, (millis));        \
    }                                                                      \
  }
#endif /* USDT2 */
```

The second problem is the use of lower case probe names in the code. For example, the following probe takes a type argument that is encorporated into the probe name with preprocessor token concatenation.

```
#define DTRACE_CLASSINIT_PROBE(type, clss, thread_type)          \
  {                                                              \
    char* data = NULL;                                           \
    int len = 0;                                                 \
    Symbol* name = (clss)->name();                               \
    if (name != NULL) {                                          \
      data = (char*)name->bytes();                               \
      len = name->utf8_length();                                 \
    }                                                            \
    HS_DTRACE_PROBE4(hotspot, class__initialization__##type,     \
      data, len, (clss)->class_loader(), thread_type);           \
  }
#endif /* USDT1 */
```

Since the USDT2 probe macros are automatically generated in upper case, the USDT2 code uses macros to translate the lower case type names into the required upper case names:

```
#ifdef USDT2
#define HOTSPOT_CLASS_INITIALIZATION_required HOTSPOT_CLASS_INITIALIZATION_REQUIRED
#define HOTSPOT_CLASS_INITIALIZATION_recursive HOTSPOT_CLASS_INITIALIZATION_RECURSIVE
#define HOTSPOT_CLASS_INITIALIZATION_concurrent HOTSPOT_CLASS_INITIALIZATION_CONCURRENT
#define HOTSPOT_CLASS_INITIALIZATION_erroneous HOTSPOT_CLASS_INITIALIZATION_ERRONEOUS
#define HOTSPOT_CLASS_INITIALIZATION_super__failed HOTSPOT_CLASS_INITIALIZATION_SUPER_FAILED
#define HOTSPOT_CLASS_INITIALIZATION_clinit HOTSPOT_CLASS_INITIALIZATION_CLINIT
#define HOTSPOT_CLASS_INITIALIZATION_error HOTSPOT_CLASS_INITIALIZATION_ERROR
#define HOTSPOT_CLASS_INITIALIZATION_end HOTSPOT_CLASS_INITIALIZATION_END
#define DTRACE_CLASSINIT_PROBE(type, clss, thread_type)          \
  {                                                              \
    char* data = NULL;                                           \
    int len = 0;                                                 \
    Symbol* name = (clss)->name();                               \
    if (name != NULL) {                                          \
      data = (char*)name->bytes();                               \
      len = name->utf8_length();                                 \
    }                                                            \
    HOTSPOT_CLASS_INITIALIZATION_##type(                         \
      data, len, (clss)->class_loader(), thread_type);           \
  }
```