```
# $kbyanc: dyntrace/dyntrace/Makefile,v 1.7 2004/12/23 01:45:19 kbyanc Exp $

PROG=           dyntrace

PREFIX?=        /usr/local
BINDIR=         ${PREFIX}/bin
MANDIR?=        ${PREFIX}/man/man

SRCS=           log.c \
                main.c \
                optree.c \
                procfs_freebsd.c \
                ptrace.c \
                radix.c \
                region.c \
                target_freebsd.c

MAN=            dyntrace.1

XML_CFLAGS!=    xml2-config --cflags
XML_LDFLAGS!=   xml2-config --libs

DEBUG_FLAGS+=   -O0 -g
CFLAGS+=        ${XML_CFLAGS}
LDFLAGS+=       ${XML_LDFLAGS}

# Flags for enabling PMC support
#CFLAGS+=       -DHAVE_PMC=1
#LDADD+=        -lpmc

NO_WERROR=      yes

WARNS?=         6

.include <bsd.prog.mk>
```

```
/*
 * Copyright (c) 2004 Kelly Yancey
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $kbyanc: dyntrace/dyntrace/dyntrace.h,v 1.13 2004/12/23 01:45:19 kbyanc Exp $
 */

#ifndef _INCLUDE_DYNTRACE_H
#define _INCLUDE_DYNTRACE_H

#include <sys/cdefs.h>
#include <stdbool.h>
#include <stdio.h>

#ifndef __GNUC__
#define __attribute__()
#endif

#undef __DECONST
#define __DECONST(type, var)    ((type)(uintptr_t)(const void *)(var))


typedef struct target_state *target_t;


typedef enum {
        REGION_UNKNOWN          = 0,
        REGION_TEXT_UNKNOWN     = 1,
        REGION_TEXT_PROGRAM     = 2,
        REGION_TEXT_LIBRARY     = 3,
        REGION_NONTEXT_UNKNOWN  = 4,
        REGION_DATA             = 5,
        REGION_STACK            = 6
} region_type_t;

#define NUMREGIONTYPES            7
#define REGION_IS_TEXT(rt)      ((rt) < REGION_NONTEXT_UNKNOWN)

typedef struct region_info *region_t;
typedef struct region_list *region_list_t;

extern const char *region_type_name[NUMREGIONTYPES];
```

```
extern bool      opt_debug;
extern bool      opt_printzero;
extern char     *opt_outfile;

#define debug(fmt, ...) do {                    \
        if (opt_debug) warn(fmt, __VA_ARGS__);  \
} while (0)


__BEGIN_DECLS
extern void      warn(const char *fmt, ...)
                       __attribute__ ((format (printf, 1, 2)));
extern void      fatal(int eval, const char *fmt, ...)
                      __attribute__ ((noreturn, format (printf, 2, 3)));


extern void      setsighandler(int sig, void (*handler)(int));


extern region_list_t
                 region_list_new(void);
extern void      region_list_done(region_list_t *rlistp);

extern region_t  region_lookup(region_list_t rlist, vm_offset_t addr);
extern void      region_update(region_list_t rlist,
                               vm_offset_t start, vm_offset_t end,
                               region_type_t type, bool readonly);
extern size_t    region_read(target_t targ, region_t region,
                             vm_offset_t offset, void *dest, size_t len);
extern region_type_t
                 region_get_type(region_t region);
extern size_t    region_get_range(region_t region,
                                  vm_offset_t *startp, vm_offset_t *endp);


extern void      optree_parsefile(const char *filepath);
extern void      optree_update(target_t targ, region_t region,
                               vm_offset_t pc, uint cycles);
extern void      optree_output_open(void);
extern void      optree_output(void);


extern void      target_init(void);
extern void      target_done(void);

extern target_t  target_execvp(const char *path, char * const argv[]);
extern target_t  target_attach(pid_t pid);
extern void      target_detach(target_t *targp);

extern target_t  target_wait(void);
extern void      target_step(target_t targ);

extern size_t    target_read(target_t targ, vm_offset_t addr,
                             void *dest, size_t len);

extern vm_offset_t target_get_pc(target_t targ);
extern uint      target_get_cycles(target_t targ);
extern const char *target_get_name(target_t targ);
extern region_t  target_get_region(target_t targ, vm_offset_t offset);

__END_DECLS

#endif
```

```
/*
 * Copyright (c) 2004 Kelly Yancey
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $kbyanc: dyntrace/dyntrace/log.c,v 1.4 2004/12/27 10:23:30 kbyanc Exp $
 */

#include <assert.h>
#include <errno.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "dyntrace.h"


#define WARN_BUFFER_SIZE        128


static void      expand_error(const char *src, char *dest, size_t destlen,
                              int errnum);
static void      warnv(const char *fmt, va_list ap);


/*!
 * expand_error() - Perform syslog(3)-like expansion of \%m to error message.
 *
 *      Copies all text from \a src to \a dest, replacing any instances of
 *      \%m in the source text with the error message returned by strerror(3)
 *      for \a errnum.  Guarantees that no more than \a destlen bytes are
 *      written to the \a dest buffer.  Destination string is always
 *      nul-terminated on return, even if it was truncated.
 *
 *      @param  src             Source text to copy.
 *
 *      @param  dest            Destination buffer to write expanded text to.
 *
 *      @param  destlen         Size of destination buffer.
 *
 *      @param  errnum          Error number to use to lookup error message
 *                              to replace \%m with.
 */
```

```
void
expand_error(const char *src, char *dest, size_t destlen, int errnum)
{
        const char *errstr;
        const char *m;
        size_t len;

        assert(destlen > 0);
        destlen--;              /* Ensure room to nul-terminate string. */

        while (*src != '\0' && destlen > 0) {

                m = strstr(src, "%m");
                if (m == NULL) {
                        strncpy(dest, src, destlen);
                        return;
                }

                /*
                 * Append text preceeding the '%m' marker.
                 */
                len = m - src;
                if (len > destlen)
                        len = destlen;
                memcpy(dest, src, len);
                destlen -= len;
                dest += len;

                /*
                 * Lookup the error message to replace the '%m' with.
                 */
                errstr = strerror(errnum);
                if (errstr == NULL)
                        errstr = "unknown error";

                /*
                 * Append error message text.
                 */
                len = strlen(errstr);
                if (len > destlen)
                        len = destlen;
                memcpy(dest, errstr, len);
                destlen -= len;
                dest += len;

                src = m + 2;
        }

        *dest = '\0';
}


/*!
 * warnx() - Write warning to stderr with %m expanded to error message.
 *
 *      @param  fmt             printf(3)-style format specifier indicating
 *                              how to format the output.
 *
 *      @param  ap              stdarg(3) variable-length arguments.
 */
void
warnv(const char *fmt, va_list ap)
{
        static char fmtbuf[WARN_BUFFER_SIZE];
        int saved_errno;
```

```
        const char *nl;
        const char *m;

        saved_errno = errno;

        assert(fmt != NULL);

        m = strstr(fmt, "%m");
        if (m != NULL) {
                expand_error(fmt, fmtbuf, sizeof(fmtbuf), saved_errno);
                fmt = fmtbuf;
        }

        vfprintf(stderr, fmt, ap);

        /*
         * Append a trailing newline if one is not supplied.
         */
        nl = strrchr(fmt, '\n');
        if (nl == NULL || nl[1] != '\0')
                fputc('\n', stderr);
}


/*!
 * warn() - Display warning.
 *
 *      Writes a formatted error message to the standard error output and
 *      returns.
 */
void
warn(const char *fmt, ...)
{
        va_list ap;

        va_start(ap, fmt);
        warnv(fmt, ap);
        va_end(ap);
}


/*!
 * fatal() - Report a fatal error and exit.
 *
 *      Writes a formatted error message to the standard error output before
 *      exiting with the given exit code.
 */
void
fatal(int eval, const char *fmt, ...)
{
        va_list ap;

        va_start(ap, fmt);
        warnv(fmt, ap);
        va_end(ap);

        exit(eval);
}
```

```
/*
 * Copyright (c) 2004 Kelly Yancey
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $kbyanc: dyntrace/dyntrace/main.c,v 1.16 2004/12/23 01:45:19 kbyanc Exp $
 */

#include <sys/types.h>
#include <sys/time.h>

#include <assert.h>
#include <inttypes.h>
#include <libgen.h>
#include <signal.h>
#include <stdlib.h>
#include <sysexits.h>
#include <time.h>
#include <unistd.h>

#include "dyntrace.h"

#define DEFAULT_CHECKPOINT      (15 * 60)       /* 15 minutes */
#define DEFAULT_OPFILE          "/usr/local/share/dyntrace/oplist-x86.xml"


static void      usage(const char *msg);
static void      trace(target_t targ);
static void      time_record(const char *msg, struct timeval *tvp);
static void      epilogue(void);
static uint      rounddiv(uint64_t a, uint64_t b);
static void      sig_terminate(int sig);
static void      sig_checkpoint(int sig);


static struct timeval starttime, stoptime;
static uint64_t  instructions   = 0;

static volatile sig_atomic_t terminate  = false;
static volatile sig_atomic_t checkpoint = false;

       bool      opt_debug      = false;
       bool      opt_printzero  = false;
       int       opt_checkpoint = -1;
static pid_t     opt_pid        = -1;
```

```
	char	*opt_outfile	= NULL;
	char	*opt_command	= NULL;


void
usage(const char *msg)
{
	const char *progname;

	if (msg != NULL)
		warn("%s\n", msg);

	progname = getprogname();

	fatal(EX_USAGE,
"usage: %s [-vz] [-c seconds] [-f opcodefile] [-o outputfile] command\n"
"       %s [-vz] [-c seconds] [-f opcodefile] [-o outputfile] -p pid\n",
		progname, progname
	);
}


int
main(int argc, char *argv[])
{
	bool opsloaded = false;
	target_t targ;
	int ch;

	if (argc == 1)
		usage(NULL);

	while ((ch = getopt(argc, argv, "c:f:o:p:vz")) != -1) {
		switch ((char)ch) {
		case 'c':
			opt_checkpoint = atoi(optarg);
			if (opt_checkpoint < 0) {
				fatal(EX_USAGE, "invalid count for -c: \"%s\"",
					optarg);
			}
			break;

		case 'f':
			optree_parsefile(optarg);
			opsloaded = true;
			break;

		case 'o':
			if (opt_outfile != NULL)
				usage("only one output file can be specified");
			opt_outfile = optarg;
			break;

		case 'p':
			if (opt_pid != -1)
				usage("only one process id can be specified");
			opt_pid = atoi(optarg);
			if (opt_pid <= 0) {
				fatal(EX_USAGE,
					"expected process id, got \"%s\"",
					optarg);
			}
			break;

		case 'v':
```

```
                        opt_debug = true;
                        break;

                case 'z':
                        opt_printzero = true;
                        break;

                case '?':
                default:
                        usage(NULL);
                }
        }

        argv += optind;
        argc -= optind;

        if (opt_checkpoint == -1)
                opt_checkpoint = DEFAULT_CHECKPOINT;
        if (!opsloaded) {
                optree_parsefile(DEFAULT_OPFILE);
                opsloaded = true;
        }

        target_init();

        if (opt_pid != -1) {
                if (argc != 0)
                        usage("cannot specify both a process id and a command");

                targ = target_attach(opt_pid);
        }
        else {
                if (argc == 0)
                        usage("command not specified");

                targ = target_execvp(*argv, argv);
        }

        if (opt_outfile == NULL)
                asprintf(&opt_outfile, "%s.trace", target_get_name(targ));

        optree_output_open();
        warn("recording results to %s", opt_outfile);

        /*
         * Install signal handlers to ensure we dump the collected data
         * before terminating.
         */
        setsighandler(SIGHUP, sig_terminate);
        setsighandler(SIGINT, sig_terminate);
        setsighandler(SIGQUIT, sig_terminate);
        setsighandler(SIGTERM, sig_terminate);

        /*
         * Install signal handlers to dump collected data on demand.  This
         * is used to implement periodic checkpointing (via SIGALRM) and to
         * allow external programs to request updates (via SIGUSR1 or SIGINFO).
         */
        setsighandler(SIGALRM, sig_checkpoint);
        setsighandler(SIGUSR1, sig_checkpoint);
#ifdef SIGINFO
        setsighandler(SIGINFO, sig_checkpoint);
#endif

        if (opt_checkpoint == 0)
```

```
                warn("checkpoints disabled");
        else {
                alarm(opt_checkpoint);
                warn("checkpoints every %u seconds",
                     opt_checkpoint);
        }

        time_record("trace started at", &starttime);

        trace(targ);

        time_record("trace stopped at", &stoptime);
        epilogue();

        optree_output();

        /*
         * If we attached to an already running process (i.e. -p pid command
         * line option was used) and that process has not terminated, then
         * detach from it so it can continue running like it was before we
         * started tracing it.
         *
         * However, if the traced process is our child process, do not
         * detach from it if it is still running so that it is killed when
         * we exit.
         */
        if (terminate && opt_pid > 0)
                target_detach(&targ);

        target_done();

        return 0;
}


void
trace(target_t targ)
{

        while (!terminate) {
                vm_offset_t pc = target_get_pc(targ);
                region_t region = target_get_region(targ, pc);
                uint cycles = target_get_cycles(targ);

                optree_update(targ, region, pc, cycles);
                instructions++;

                /*
                 * Periodically record the instruction counters in case
                 * we get interrupted (e.g. power outage, etc) so at least
                 * we have something to show for our efforts.
                 */
                if (checkpoint) {
                        warn("checkpoint");
                        optree_output();
                        optree_output_open();
                        checkpoint = false;
                }

                if (terminate)
                        break;

                target_step(targ);
                targ = target_wait();
                if (targ == NULL)
```

```
                        break;
        }
}


void
time_record(const char *msg, struct timeval *tvp)
{
        char timestr[64];
        time_t seconds;

        gettimeofday(tvp, NULL);
        seconds = tvp->tv_sec;

        if (opt_debug) {
                strftime(timestr, sizeof(timestr), "%c", localtime(&seconds));
                debug("=== %s %s ===", msg, timestr);
        }
}


void
epilogue(void)
{
        uint ips;

        if (!opt_debug)
                return;

        stoptime.tv_sec -= starttime.tv_sec;
        stoptime.tv_usec -= starttime.tv_usec;
        if (stoptime.tv_usec < 0) {
                stoptime.tv_usec += 1000000;
                stoptime.tv_sec--;
        }

        ips = rounddiv(instructions * 1000000,
                        (stoptime.tv_sec * 1000) +
                        rounddiv(stoptime.tv_usec, 1000));
        debug("%llu instructions traced in "
              "%0lu.%03u seconds (%0u.%03u/sec)",
              (unsigned long long)instructions,
              stoptime.tv_sec, rounddiv(stoptime.tv_usec, 1000),
              ips / 1000, ips % 1000);
}




uint
rounddiv(uint64_t a, uint64_t b)
{
        return (a + (b / 2)) / b;
}


void
setsighandler(int sig, void (*handler)(int))
{
        struct sigaction act;

        act.sa_handler = handler;
        act.sa_flags = SA_RESTART;
        sigemptyset(&act.sa_mask);
        sigaction(sig, &act, NULL);
```

```
}


void
sig_terminate(int sig __unused)
{
        terminate = true;
}


void
sig_checkpoint(int sig)
{
        checkpoint = true;
        if (sig == SIGALRM && opt_checkpoint > 0)
                alarm(opt_checkpoint);
}
```

```
/*
 * Copyright (c) 2004 Kelly Yancey
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $kbyanc: dyntrace/dyntrace/optree.c,v 1.14 2004/12/27 10:31:35 kbyanc Exp $
 */

#include <libxml/xmlreader.h>
#include <libxml/xmlwriter.h>

#include <assert.h>
#include <ctype.h>
#include <fcntl.h>
#include <inttypes.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <sysexits.h>
#include <unistd.h>

#if defined(__FreeBSD__) && __FreeBSD__ >= 5
#include <arpa/inet.h>  /* for htonl() */
#endif

#include "dyntrace.h"
#include "radix.h"

/*!
 * @file
 *
 * We use the same radix tree code that FreeBSD (and other 4.4BSD derivatives)
 * uses for routing lookups to implement opcode identification.  This data
 * structure is perfectly suited for matching opcode bit strings as it provides
 * best-match lookups with masking.  Masking is especially useful as it allows
 * for don't-care bits in opcode bit strings (a requirement for the x86
 * instruction set and possibly others).
 *
 * For an explanation of how the radix tree works, see:
 * Gary R. Wright and W. Richard Stevens. TCP/IP Illustrated, Volume 2:
 * The Implementation, chapter 18.
 *
 */
```

```
/*!
 * @struct bitval
 *
 * Data structure representing a string of bits up to 32 bits long.  Used as a
 * key for radix tree lookups so the first byte must include the length of the
 * structure in bytes (simulating a BSD sockaddr structure for which the radix
 * code was originally designed).  Furthermore, the bit string itself is
 * aligned on a word boundary to improve performance.  There will likely be
 * compiler-added padding between the len and val members.
 *
 *      @param  len             Length of the structure in bytes.
 *                              Same as sizeof(struct bitval).
 *
 *      @param  val             Storage for bit string.  Aligned to word
 *                              boundary to allow for fast word-sized access.
 */
struct bitval {
        uint8_t          len;
        uint32_t         val;           /* XXX Should be uint_fast32_t. */
};


/*!
 * @struct OpTreeNode
 *
 * Data structure representing a single opcode.  This is used as an entry in
 * the radix tree so the first 2 fields must be pointers to radix tree nodes
 * (simulating a BSD rtentry structure).
 */
struct OpTreeNode {
        struct radix_node rn[2];
        struct bitval    match;
        struct bitval    mask;
        enum { OPCODE, PREFIX } type;
};


typedef uint prefixmask_t;
#define PREFIXMASK_EMPTY        0
#define MAX_PREFIXES            (sizeof(prefixmask_t) * 8)

struct Prefix {
        struct OpTreeNode node;

        uint8_t          len;
        uint8_t          id;
        prefixmask_t     mask;
        char            *bitmask;
        char            *detail;
};


/*!
 * @struct counter
 *
 *      Each opcode has a list of counters per memory region type.  Each
 *      counter in the list represents the usage count and timing for the
 *      opcode with a given set of prefixes.  Since the most common case
 *      is an opcode unadorned with prefix bytes, the first counter in the
 *      list is embedded within the opcode structure itself and has a nul
 *      prefix mask.
 *
 *      @param  next            Pointer to next counter in list.
 *
```

```
 *      @param  prefixmask      Prefix mask this counter is for.
 *
 *      @param  count           The number of times the opcode has been
 *                              executed with our list of prefixes.
 *
 *      @param  cycles_total    The total number of CPU cycles accumulated
 *                              across all executions.
 *
 *      @param  cycles_min      The minimum number of CPU cycles for any
 *                              single execution.
 *
 *      @param  cycles_max      The maximum number of CPU cycles for any
 *                              single execution.
 */
struct counter {
        struct counter  *next;
        prefixmask_t     prefixmask;

        uint64_t         n;
        uint64_t         cycles_total;
        uint             cycles_min;
        uint             cycles_max;
};


/*!
 * @struct Opcode
 */
struct Opcode {
        struct OpTreeNode node;

        struct counter   count_head[NUMREGIONTYPES];
        struct counter  *count_end[NUMREGIONTYPES];

        char            *bitmask;
        char            *mnemonic;
        char            *detail;
};


static struct radix_node_head *op_rnh = NULL;
static struct Prefix prefix_index[MAX_PREFIXES];
static uint      prefix_count = 0;
static xmlTextWriterPtr writer = NULL;
static int       writer_fd = -1;
static bool      region_type_use[NUMREGIONTYPES];


static void      optree_init(void);
static bool      optree_insert(struct OpTreeNode *op);
static struct OpTreeNode *optree_lookup(const void *keyptr);
static int       optree_print_node(struct radix_node *rn, void *arg);

static struct Opcode *opcode_alloc(void);
static void      opcode_parse(xmlNode *node);
static void      opcode_free(struct Opcode **opp);

static const char *prefix_string(prefixmask_t prefixmask);
static void      prefix_parse(xmlNode *node);
static void      prefix_free(struct Prefix **prefixp);

static void      parse_bitmask(const char *bitstr,
                               uint32_t *mask, uint32_t *match);
```

```
/*!
 * optree_init() - Initialize radix tree routines for use as opcode lookup tree.
 */
void
optree_init(void)
{
        struct Opcode *op;

        assert(op_rnh == NULL);

        /*
         * Set the maximum key length and initialize the radix tree library.
         * Tell rn_inithead() at which byte offset to find significant key
         * bits.
         */
        max_keylen = sizeof(struct bitval);
        rn_init();
        rn_inithead((void **)&op_rnh, offsetof(struct bitval, val));

        assert(op_rnh != NULL);

        /*
         * Add a catch-all default opcode entry.
         */
        op = opcode_alloc();
        op->bitmask = strdup("");
        op->mnemonic = strdup("(unknown)");
        op->detail = NULL;
        op->node.match.len = op->node.mask.len = 0;
        op_rnh->rnh_addaddr(&op->node.match, &op->node.mask, op_rnh,
                            (void *)op);

        /* Clear our per-region use flags. */
        memset(region_type_use, 0, sizeof(region_type_use));
}


/*!
 * optree_insert() - Add opcode to tree.
 *
 *      @param  node            Pointer to node to add to tree.
 *
 *      @return Boolean indicating whether or not the opcode was successfully
 *              added to the tree.
 */
bool
optree_insert(struct OpTreeNode *node)
{
        struct radix_node *rn;
        struct OpTreeNode *xnode;

        assert(node->match.len == sizeof(node->match) &&
               node->mask.len == sizeof(node->mask));

        rn = op_rnh->rnh_addaddr(&node->match, &node->mask, op_rnh,
                                 (void *)node);
        if (rn != NULL)
                return true;

        /*
         * If we were unable to add the new entry, then another node with
         * the same bitmask must already exist in the tree.  Find out what
         * node it is so we can inform the user.
         */
```

```
        xnode = optree_lookup(&node->match.val);
        assert (xnode != NULL && xnode != node);

#ifdef XXXX
//      if (strcmp(xop->mnemonic, op->mnemonic) != 0) {
                warn("opcodes %s and %s have the same bitmask \"%s\"",
                     op->mnemonic, xop->mnemonic, op->bitmask);
//      }
#endif

        return false;
}


/*!
 * optree_lookup() - Lookup opcode.
 *
 *
 *
 *      @param  keyptr          Pointer to XXX.
 *
 */
struct OpTreeNode *
optree_lookup(const void *keyptr)
{
        struct bitval key;
        struct OpTreeNode *op;

        key.len = sizeof(key);
        memcpy(&key.val, keyptr, sizeof(key.val));

        op = (struct OpTreeNode *)op_rnh->rnh_lookup(&key, NULL, op_rnh);
        return op;
}


void
optree_update(target_t targ, region_t region, vm_offset_t pc, uint cycles)
{
        struct OpTreeNode *node;
        struct Prefix *prefix;
        struct Opcode *op;
        struct counter *c;
        region_type_t regiontype;
        prefixmask_t prefixmask = PREFIXMASK_EMPTY;
        uint32_t text;

        assert(region != NULL);

        regiontype = region_get_type(region);
        assert(regiontype < NUMREGIONTYPES);

        region_type_use[regiontype] = true;

        /*
         * First, build mask of all prefixes before the opcode.
         */
        for (;;) {
                text = 0;
                region_read(targ, region, pc, &text, sizeof(text));

                node = optree_lookup(&text);
                assert(node != NULL);
                if (node->type != PREFIX)
```

```
                        break;

                prefix = (struct Prefix *)node;

                pc += prefix->len;
                prefixmask |= prefix->mask;
        }

        assert(node->type == OPCODE);
        op = (struct Opcode *)node;

        /*
         * Locate the counter to update by its prefix mask.
         */
        for (c = &op->count_head[regiontype]; c != NULL; c = c->next) {
                if (c->prefixmask == prefixmask)
                        break;
        }

        /*
         * If there is no existing counter for the current prefix mask,
         * append a new counter to the end of the list.
         */
        if (c == NULL) {
                c = calloc(1, sizeof(*c));
                if (c == NULL)
                        fatal(EX_OSERR, "malloc: %m");
                op->count_end[regiontype]->next = c;
                c->next = NULL;
                c->prefixmask = prefixmask;
        }

        c->n++;
        c->cycles_total += cycles;
        if (cycles < c->cycles_min)
                c->cycles_min = cycles;
        else if (cycles > c->cycles_max)
                c->cycles_max = cycles;

        /*
         * Warn about instructions which match the default opcode.
         * In order to reduce verbosity, we only print the warning when
         * the current program counter differs from the last program counter
         * at which we found an unknown opcode.
         */
        if (op->node.match.len == 0) {
                static vm_offset_t prevpc = 0;
                if (pc != prevpc) {
                        warn("unknown opcode at pc 0x%08x: 0x%08x", pc, text);
                        prevpc = pc;
                }
        }
}


void
optree_output_open(void)
{
        xmlOutputBufferPtr out;

        assert(opt_outfile != NULL);
        assert(writer == NULL);

        /*
         * Open the output file for writing.  We keep the output file open
```

```
         * across multiple calls, overwriting the contents of the file each
         * time we are called (e.g. checkpointing).  We only truncate the
         * output file when we first open the file, after that the file can
         * only get longer each time we write it as we either find new
         * instructions or the instruction counts grow.
         */
        if (writer_fd < 0) {
                writer_fd = open(opt_outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666);
                if (writer_fd < 0) {
                        fatal(EX_OSERR, "unable to open %s for writing: %m",
                              opt_outfile);
                }
        }

        lseek(writer_fd, 0, SEEK_SET);

        out = xmlOutputBufferCreateFd(writer_fd, NULL);
        if (out == NULL) {
                fatal(EX_CANTCREAT, "unable to open %s for writing: %m",
                      opt_outfile);
        }

        writer = xmlNewTextWriter(out);
        if (writer == NULL) {
                xmlOutputBufferClose(out);
                fatal(EX_CANTCREAT, "unable to open %s for writing: %m",
                      opt_outfile);
        }

        xmlTextWriterSetIndent(writer, 4);
}


void
optree_output(void)
{
        const struct Prefix *prefix;
        region_type_t regiontype;
        uint i;

        assert(writer != NULL);

        if (xmlTextWriterStartDocument(writer, NULL, "utf-8", NULL) < 0)
                fatal(EX_IOERR, "failed to write to %s: %m", opt_outfile);

        xmlTextWriterStartElement(writer, "dyntrace");

        /* First, output a list of prefixes. */
        for (i = 0; i < prefix_count; i++) {
                prefix = &prefix_index[i];
                xmlTextWriterStartElement(writer, "prefix");
                xmlTextWriterWriteAttribute(writer, "id",
                                            prefix_string(prefix->mask));
                xmlTextWriterWriteAttribute(writer, "bitmask", prefix->bitmask);
                xmlTextWriterWriteAttribute(writer, "detail", prefix->detail);
                xmlTextWriterEndElement(writer /* prefix */);
        }

        xmlTextWriterStartElement(writer, "program");
        xmlTextWriterWriteAttribute(writer, "name", "N/A");     /* XXX */

        /*
         * Iterate through the region types, outputting the opcodes in each
         * region.
         */
```

```
        for (regiontype = 0; regiontype < NUMREGIONTYPES; regiontype++) {

                if (!region_type_use[regiontype])
                        continue;

                xmlTextWriterStartElement(writer, "region");
                xmlTextWriterWriteAttribute(writer, "type",
                                        region_type_name[regiontype]);

                op_rnh->rnh_walktree(op_rnh, optree_print_node, &regiontype);

                xmlTextWriterEndElement(writer /* "region */);
        }

        xmlTextWriterEndElement(writer /* "program" */);
        xmlTextWriterEndElement(writer /* "dyntrace" */);
        xmlTextWriterEndDocument(writer);
        xmlFreeTextWriter(writer);

        writer = NULL;

        /* Ensure the results are written to disk. */
        fsync(writer_fd);
}


const char *
prefix_string(prefixmask_t prefixmask)
{
        static char buffer[100];
        size_t len;
        prefixmask_t checkmask;
        int id;

        /* No instruction prefix is the most common case. */
        if (prefixmask == 0)
                return "";

        len = 0;

        for (id = 0, checkmask = 1; prefixmask != 0; id++, checkmask <<= 1) {
                char idstr[3] = { 'A', '\0', '\0' };
                size_t idstrlen = 1;

                if ((prefixmask & checkmask) == 0)
                        continue;
                prefixmask &= ~checkmask;

                if (id < 26)
                        idstr[0] += id;
                else {
                        idstr[1] = 'A' + id - 26;
                        idstrlen++;
                }

                assert(len + idstrlen + 1 < sizeof(buffer));
                assert(idstrlen <= 2);

                if (len > 0)
                        buffer[len++] = ',';

                buffer[len + 0] = idstr[0];
                buffer[len + 1] = idstr[1];
                len += idstrlen;
        }
```

```
        buffer[len] = '\0';
        return buffer;
}


int
optree_print_node(struct radix_node *rn, void *arg)
{
        const struct OpTreeNode *node = (struct OpTreeNode *)rn;
        const struct Opcode *op = (const struct Opcode *)node;
        region_type_t regiontype = *(const region_type_t *)arg;
        const struct counter *c;
        char buffer[32];

        if (node->type != OPCODE)
                return 0;

        for (c = &op->count_head[regiontype]; c != NULL; c = c->next) {

                /*
                 * Skip counters with zero counts unless the printzero option
                 * was specified on the command line.
                 */
                if (c->n == 0 && !opt_printzero)
                        continue;

                xmlTextWriterStartElement(writer, "opcount");
                xmlTextWriterWriteAttribute(writer, "bitmask", op->bitmask);
                xmlTextWriterWriteAttribute(writer, "mnemonic", op->mnemonic);
                if (op->detail != NULL) {
                        xmlTextWriterWriteAttribute(writer, "detail",
                                                    op->detail);
                }

                if (c->prefixmask != 0) {
                        xmlTextWriterWriteAttribute(writer, "prefixes",
                                        prefix_string(c->prefixmask));
                }

                snprintf(buffer, sizeof(buffer), "%llu",
                         (unsigned long long)c->n);
                xmlTextWriterWriteAttribute(writer, "n", buffer);

                /* Only output cycle counts if we have them. */
                if (c->cycles_total == 0) {
                        xmlTextWriterEndElement(writer /* "opcount" */);
                        continue;
                }

                snprintf(buffer, sizeof(buffer), "%llu",
                         (unsigned long long)c->cycles_total);
                xmlTextWriterWriteAttribute(writer, "cycles", buffer);

                snprintf(buffer, sizeof(buffer), "%u", c->cycles_min);
                xmlTextWriterWriteAttribute(writer, "min", buffer);

                snprintf(buffer, sizeof(buffer), "%u", c->cycles_min);
                xmlTextWriterWriteAttribute(writer, "max", buffer);

                xmlTextWriterEndElement(writer /* "opcount" */);
        }

        return 0;
}
```

```
void
optree_parsefile(const char *filepath)
{
        xmlTextReaderPtr reader;
        int ret;

        if (op_rnh == NULL)
                optree_init();

        LIBXML_TEST_VERSION

        reader = xmlNewTextReaderFilename(filepath);
        if (reader == NULL)
                fatal(EX_NOINPUT, "unable to open %s for reading", filepath);

        while ((ret = xmlTextReaderRead(reader)) > 0) {
                xmlNode *node;

                if (xmlTextReaderNodeType(reader) != XML_ELEMENT_NODE)
                        continue;

                node = xmlTextReaderExpand(reader);

                if (strcmp(node->name, "prefix") == 0)
                        prefix_parse(node);

                if (strcmp(node->name, "op") == 0)
                        opcode_parse(node);
        }

        if (ret != 0)
                fatal(EX_DATAERR, "failed to parse %s", filepath);

        xmlFreeTextReader(reader);
}



void
opcode_parse(xmlNode *node)
{
        const xmlAttr *attr;
        struct Opcode *op;

        op = opcode_alloc();

        for (attr = node->properties; attr != NULL; attr = attr->next) {
                const char *name = attr->name;
                const char *value = XML_GET_CONTENT(attr->children);

                if (strcmp(name, "bitmask") == 0)
                        op->bitmask = strdup(value);
                else if (strcmp(name, "mnemonic") == 0)
                        op->mnemonic = strdup(value);
                else if (strcmp(name, "detail") == 0)
                        op->detail = strdup(value);
        }

        /*
         * Verify the opcode looks complete.
         */
        if (op->bitmask == NULL) {
                fatal(EX_DATAERR, "bitmask missing at %ld",
```

```
                        XML_GET_LINE(node));
        }
        if (op->mnemonic == NULL) {
                fatal(EX_DATAERR, "mnemonic missing at %ld",
                        XML_GET_LINE(node));
        }

        parse_bitmask(op->bitmask, &op->node.mask.val, &op->node.match.val);

        if (!optree_insert(&op->node)) {
                opcode_free(&op);
        }
}


struct Opcode *
opcode_alloc(void)
{
        struct Opcode *op;
        region_type_t regiontype;

        op = calloc(1, sizeof(*op));
        if (op == NULL)
                fatal(EX_OSERR, "malloc: %m");

        for (regiontype = 0; regiontype < NUMREGIONTYPES; regiontype++)
                op->count_end[regiontype] = &op->count_head[regiontype];

        op->node.type = OPCODE;
        op->node.mask.len = sizeof(op->node.mask);
        op->node.match.len = sizeof(op->node.match);

        return op;
}


void
opcode_free(struct Opcode **opp)
{
        struct Opcode *op = *opp;

        *opp = NULL;
        if (op->bitmask != NULL)
                free(op->bitmask);
        if (op->mnemonic != NULL)
                free(op->mnemonic);
        if (op->detail != NULL)
                free(op->detail);
        free(op);
}


void
prefix_parse(xmlNode *node)
{
        const xmlAttr *attr;
        struct Prefix *prefix;

        if (prefix_count >= MAX_PREFIXES) {
                fatal(EX_SOFTWARE, "cannot specify more than %u prefixes",
                        MAX_PREFIXES);
        }

        prefix = &prefix_index[prefix_count];
```

```
        for (attr = node->properties; attr != NULL; attr = attr->next) {
                const char *name = attr->name;
                const char *value = XML_GET_CONTENT(attr->children);

                if (strcmp(name, "bitmask") == 0)
                        prefix->bitmask = strdup(value);
                else if (strcmp(name, "detail") == 0)
                        prefix->detail = strdup(value);
        }

        /*
         * Verify the prefix looks complete.
         */
        if (prefix->bitmask == NULL) {
                fatal(EX_DATAERR, "bitmask missing at %ld",
                      XML_GET_LINE(node));
        }

        prefix->node.type = PREFIX;
        prefix->node.mask.len = sizeof(prefix->node.mask);
        prefix->node.match.len = sizeof(prefix->node.match);

        parse_bitmask(prefix->bitmask,
                      &prefix->node.mask.val, &prefix->node.match.val);

        if (!optree_insert(&prefix->node)) {
                prefix_free(&prefix);
                return;
        }

        prefix->len = (strlen(prefix->bitmask) + 7) / 8;
        prefix->id = prefix_count;
        prefix->mask = 1 << prefix_count;
        prefix_count++;
}


void
prefix_free(struct Prefix **prefixp)
{
        struct Prefix *prefix = *prefixp;

        *prefixp = NULL;

        if (prefix->bitmask != NULL)
                free(prefix->bitmask);
        if (prefix->detail != NULL)
                free(prefix->detail);
}


void
parse_bitmask(const char *bitstr, uint32_t *maskp, uint32_t *matchp)
{
        uint32_t mask;
        uint32_t match;
        uint32_t i;

        mask = match = 0;
        i = 1 << ((sizeof(i) * 8) - 1);         /* Set high bit. */

        while (*bitstr != '\0') {
                assert(i != 0);

                if (strchr("01xX", *bitstr) == NULL) {
```

```
                        fatal(EX_DATAERR,
                              "character '%c' not allowed in bitstr", *bitstr);
                }

                if (tolower(*bitstr) != 'x')
                        mask |= i;
                if (*bitstr == '1')
                        match |= i;

                i >>= 1;
                bitstr++;
        }

        /*
         * Since the opcodes are defined by a consecutive sequence of bits,
         * undo any host byte ordering.
         */
        *maskp = htonl(mask);
        *matchp = htonl(match);
}
```

```
/*
 * Copyright (c) 2004 Kelly Yancey
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $kbyanc: dyntrace/dyntrace/procfs.h,v 1.3 2004/12/23 01:45:19 kbyanc Exp $
 */

#ifndef _INCLUDE_DYNTRACE_PROCFS_H
#define _INCLUDE_DYNTRACE_PROCFS_H

#include <sys/cdefs.h>
#include <stdbool.h>


__BEGIN_DECLS

extern bool      procfs_init(void);

extern int       procfs_map_open(pid_t pid);
extern void      procfs_map_close(int *pmapfdp);
extern void      procfs_map_read(int pmapfd, void *destp, size_t *lenp);

extern int       procfs_mem_open(pid_t pid);
extern void      procfs_mem_close(int *pmemfdp);
extern size_t    procfs_mem_read(int pmemfd, vm_offset_t addr,
                                 void *dest, size_t len);

extern int       procfs_generic_open(pid_t pid, const char *node);
extern void      procfs_generic_close(int *fdp);

extern char     *procfs_get_procname(pid_t pid);

__END_DECLS

#endif
```

```
/*
 * Copyright (c) 2004 Kelly Yancey
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $kbyanc: dyntrace/dyntrace/procfs_freebsd.c,v 1.7 2004/12/27 04:31:54 kbyanc Exp $
 */

#include <sys/param.h>
#include <sys/types.h>
#include <sys/mount.h>

#include <assert.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <paths.h>
#include <regex.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <sysexits.h>
#include <unistd.h>

#include "dyntrace.h"
#include "procfs.h"


/*
 * On FreeBSD 4, the vfsconf structure exported to userland applications is
 * the same as the structure that the kernel uses.  All later versions export
 * a distinct xvfsconf to userland.
 */
#if __FreeBSD__ == 4
#define xvfsconf vfsconf
#endif


static bool      procfs_initialized = false;
static char     *procfs_path = NULL;

static bool      procfs_isavailable(void);
static bool      procfs_ismounted(char **mountpointp);
static bool      procfs_isaccessable(const char *path);
static int       procfs_opennode(const char *procfs, pid_t pid,
```

```
                                const char *node);
static bool      procfs_mount(const char *path);
static void      procfs_unmount(void);
static void      procfs_rmtmpdir(void);


/*!
 * procfs_init() - Initialize data structures for the procfs interface routines.
 *
 *      @return boolean true if procfs is available and initialized, boolean
 *              false otherwise.
 */
bool
procfs_init(void)
{

        /*
         * If procfs_init() has already been called, then procfs_path will be
         * non-null (the path where we can access procfs) if procfs is
         * available.
         */
        if (procfs_initialized)
                return procfs_path != NULL;
        procfs_initialized = true;

        /*
         * Check to see if the kernel supports procfs and if it is mounted
         * somewhere accessable.
         */
        if (!procfs_isavailable())
                return false;

        if (procfs_ismounted(&procfs_path))
                return true;

        /*
         * Procfs is available, but not already mounted.  Create a temporary
         * directory and try to mount procfs there.
         */
        procfs_path = strdup(_PATH_TMP "dyntrace.XXXXXX");
        if (mkdtemp(procfs_path) == NULL) {
                warn("failed to create directory %s to mount procfs: %m",
                     procfs_path);
                free(procfs_path);
                procfs_path = NULL;
                return false;
        }

        if (!procfs_mount(procfs_path)) {
                procfs_rmtmpdir();
                free(procfs_path);
                procfs_path = NULL;
                return false;
        }

        warn("procfs temporarily mounted on %s", procfs_path);

        /*
         * Make sure we clean up after ourselves when we are done.
         * Note that atexit() handlers are called in reverse order.
         */
        atexit(procfs_rmtmpdir);
        atexit(procfs_unmount);

        return true;
```

```
}


/*!
 * procfs_isavailable() - Determine if the procfs filesystem is supported by
 *                        the kernel.
 *
 *      If procfs is not available, tries to load the kernel module for
 *      procfs to make it available.
 *
 *      @return boolean true if the kernel supports the procfs filesystem,
 *              boolean false otherwise.
 */
bool
procfs_isavailable(void)
{
        struct xvfsconf vfc;

        /*
         * Check to see if the running kernel has support for procfs.
         */
        if (getvfsbyname("procfs", &vfc) == 0)
                return true;

#if __FreeBSD__ == 4
        /*
         * The kernel does not support procfs; try to load it as a module.
         * Only necessary for FreeBSD 4 as FreeBSD 5 and later kernels will
         * load filesystem modules automatically when they are mounted.
         * This can only succeed if the user has root privileges.
         * XXX There is no vfsunload() to unload the module when we are done.
         */
        if (!vfsisloadable("procfs") || vfsload("procfs") == 0) {
                warn("procfs is not available: %m");
                return false;
        }

        /* Deserves a warning as the system administrator may be concerned. */
        warn("loaded procfs");
#endif

        return true;
}


/*!
 * procfs_ismounted() - Determine if the procfs filesystem is mounted.
 *
 *      If procfs is not mounted, tried to mount it on a temporary directory.
 *
 *      @param  mountpointp     Pointer to populate with the address of a
 *                              newly-allocated string holding the path procfs
 *                              is mounted on.
 *
 *      @return boolean true and sets \a *mountpointp if procfs is mounted,
 *              boolean false otherwise.
 */
bool
procfs_ismounted(char **mountpointp)
{
        struct xvfsconf vfc;
        struct statfs *fsinfo;
        size_t bufsize;
        int nummounts;
        int i;
```

```
        *mountpointp = NULL;

        /*
         * Ensure the procfs filesystem is really available.
         */
        if (getvfsbyname("procfs", &vfc) != 0)
                fatal(EX_OSERR, "getvfsbyname(\"procfs\"): %m");

        /*
         * First, call getfsstat() to get the number of mounted filesystems.
         */
        nummounts = getfsstat(NULL, 0, MNT_NOWAIT);
        if (nummounts < 0)
                fatal(EX_OSERR, "getfsstat: %m");

        if (nummounts == 0)
                return false;

        /*
         * Fetch all of the mounted filesystems.  We allocate the buffer
         * one entry larger than getfsstat() said we needed just in case.
         */
        bufsize = (nummounts + 1) * sizeof(struct statfs);
        fsinfo = malloc(bufsize);
        if (fsinfo == NULL)
                fatal(EX_OSERR, "malloc: %m");

        nummounts = getfsstat(fsinfo, bufsize, MNT_NOWAIT);
        if (nummounts < 0)
                fatal(EX_OSERR, "getfsstat: %m");

        /*
         * Scan the list of mounted filesystems for a procfs filesystem we
         * have access to.  We verify access by trying to open the 'mem'
         * node corresponding to our own pid.
         */
        for (i = 0; i < nummounts; i++) {
                const struct statfs *fs = &fsinfo[i];

                if ((int)fs->f_type == vfc.vfc_typenum &&
                    procfs_isaccessable(fs->f_mntonname)) {
                        *mountpointp = strdup(fs->f_mntonname);
                        break;
                }
        }

        free(fsinfo);

        return (*mountpointp != NULL);
}


/*!
 * procfs_isaccessable() - Determine if the current process has permissions
 *                         to access the procfs filesystem mounted at the given
 *                         path.
 *
 *      @param  path    The path where procfs is mounted.
 *
 *      @return boolean true if the current process can read procfs nodes
 *              at the given path.
 */
bool
procfs_isaccessable(const char *path)
```

```
{
        int fd;

        /*
         * Test whether we can read nodes in the given procfs filesystem by
         * trying to read our own node.  This should always succeed unless
         * the filesystem is mounted on a directory with restrictive
         * permissions.
         */
        fd = procfs_opennode(path, getpid(), "mem");
        if (fd < 0)
                return false;

        close(fd);
        return true;
}


/*!
 * procfs_opennode() - Internal routine to open a procfs node for the given
 *                     process identifier.
 *
 *      @param  procfs  Path where procfs is mounted.
 *
 *      @param  pid     The process identifier whose node we are to open.
 *
 *      @param  node    The name of the procfs node (e.g. "mem", "map", etc).
 *
 *      @return file descriptor for reading from the given node.
 *
 *      The FreeBSD target only requires read access to procfs nodes, so all
 *      nodes are open by this routine read-only.
 */
int
procfs_opennode(const char *procfs, pid_t pid, const char *node)
{
        char filename[PATH_MAX];
        int fd;

        /*
         * Construct the file path to the desired procfs node.
         * Ensure that the path is nul-terminated.
         */
        snprintf(filename, sizeof(filename),
                 "%s/%u/%s", procfs, pid, node);
        filename[sizeof(filename) - 1] = '\0';

        /*
         * Open the procfs node and return the file descriptor.
         */
        fd = open(filename, O_RDONLY);
        if (fd < 0)
                fatal(EX_OSERR, "cannot open %s: %m", filename);

        return fd;
}


/*!
 * procfs_mount() - Internal routine to mount procfs at a given mount point.
 *
 *      @param  path    Path to mount procfs on.
 *
 *      @return boolean true of procfs was successfully mounted at the
 *              specified mount point path, boolean false otherwise.
```

```
 */
bool
procfs_mount(const char *path)
{

        if (mount("procfs", path, MNT_RDONLY|MNT_NOEXEC|MNT_NOSUID, NULL) < 0) {
                warn("unable to mount procfs on %s: %m", path);
                return false;
        }

        return true;
}


/*!
 * procfs_unmount() - atexit(3) handler for unmounting a temporary procfs mount.
 */
void
procfs_unmount(void)
{

        if (procfs_path == NULL)
                return;

        if (unmount(procfs_path, 0) < 0)
                warn("failed to unmount procfs from %s: %m", procfs_path);
}


/*!
 * procfs_rmtmpdir() - atexit(3) handle for removing the temporary procfs
 *                     mount point path.
 *
 *      Must be called after procfs_unmount().
 */
void
procfs_rmtmpdir(void)
{

        if (procfs_path == NULL)
                return;

        if (rmdir(procfs_path) < 0)
                warn("failed to remove %s: %m", procfs_path);

        free(procfs_path);
        procfs_path = NULL;
}


/*
 * ============================================================================
 * What follows are the implementations of the generic routines declared in
 * "procfs.h".
 * ============================================================================
 */


/*!
 * procfs_generic_open() - Open a process' procfs node.
 *
 *      @param  pid     The process identifier whose procfs node is to be
 *                      opened.
 *
 *      @param  node    Name of the procfs node to open.
```

```
 *
 *      @return file descriptor for reading from the specified node.
 *
 *      Names for procfs nodes vary from system to system; the
 *      procfs_generic_open() and procfs_generic_close() routines should only
 *      be called from system-specific code that has knowledge of the given
 *      system's node names.
 */
int
procfs_generic_open(pid_t pid, const char *node)
{

        assert(pid >= 0);

        if (!procfs_initialized)
                procfs_init();

        if (procfs_path == NULL)
                return -1;

        return procfs_opennode(procfs_path, pid, node);
}


/*!
 * procfs_generic_close() - Close a file descriptor.
 *
 *      @param  fdp     Pointer to file descriptor to close.
 *
 *      @post   The file descriptor pointed to by \a fdp is set to -1.
 */
void
procfs_generic_close(int *fdp)
{
        int fd = *fdp;

        *fdp = -1;
        if (fd >= 0)
                close(fd);
}


/*!
 * procfs_map_open() - Open process' memory-map procfs node for reading.
 *
 *      The memory-map procfs node allows the description of the given
 *      process' memory map to be read.  The exact format of the memory map
 *      is operating-system dependent.
 *
 *      @param  pid     The process identifier who memory-map node to open.
 *
 *      @return file descriptor for reading the process' memory map.
 */
int
procfs_map_open(pid_t pid)
{
        return procfs_generic_open(pid, "map");
}


/*!
 * procfs_map_close() - Close file handle for reading process' memory map.
 *
 *      @param  pmapfdp Pointer to the file descriptor to close.
 *
```

```
 *      @post   Sets the file descriptor pointed to by \a pmapfdp to -1.
 */
void
procfs_map_close(int *pmapfdp)
{
        procfs_generic_close(pmapfdp);
}


/*!
 * procfs_map_read() - Read a process's memory map.
 *
 *      @param  pmapfd  File descriptor returned by procfs_map_open() to read.
 *
 *      @param  destp   Pointer to a pointer to be populated with the address
 *                      of the memory map buffer.
 *
 *      @param  lenp    Pointer to a size_t to be populated with the number of
 *                      bytes in the memory map buffer.
 *
 *      The memory map buffer pointed to by \a destp on return is static
 *      storage and should not be freed by the caller.
 */
void
procfs_map_read(int pmapfd, void *destp, size_t *lenp)
{
        static uint8_t *buffer = NULL;
        static size_t buflen = 4096;
        uint8_t **dest = (uint8_t **)destp;
        ssize_t rv;

        assert(pmapfd >= 0);

        if (buffer == NULL) {
                buffer = malloc(buflen);
                if (buffer == NULL)
                        fatal(EX_OSERR, "malloc: %m");
        }

        /*
         * The procfs map must be read atomically.  The only way to do that
         * is to allocate a buffer large enough to read the entire text
         * at once.  Luckily, if we try to read too little, procfs fails with
         * EFBIG so we know we need to try a larger buffer.
         * XXX There should probably be a limit on how much memory we
         *     allocate.
         */
        for (;;) {
                rv = pread(pmapfd, buffer, buflen - 1, 0);

                if (rv >= 0)
                        break;                          /* Successful read. */

                if (errno != EFBIG)
                        fatal(EX_OSERR, "read: %m");    /* Unexpected error. */

                buflen <<= 1;
                buffer = realloc(buffer, buflen);
                if (buffer == NULL)
                        fatal(EX_OSERR, "realloc: %m");
        }

        buffer[rv] = '\0';
        *dest = buffer;
        *lenp = rv;
```

```
}


/*!
 * procfs_mem_open() - Open process' memory-access procfs node for reading.
 *
 *      The memory-access procfs node allows the entire virtual memory of
 *      the given process to be readable using procfs_mem_read().
 *
 *      @param  pid     Process identifier whose memory to read.
 *
 *      @return file descriptor for reading the process' memory.
 */
int
procfs_mem_open(pid_t pid)
{
        return procfs_generic_open(pid, "mem");
}


/*!
 * procfs_mem_close() - Close file descriptor for reading process' memory.
 *
 *      @param  pmemfdp Pointer to file descriptor to close.
 *
 *      @post   Sets the file descriptor pointed to by \a *pmemfdp to -1.
 */
void
procfs_mem_close(int *pmemfdp)
{
        procfs_generic_close(pmemfdp);
}


/*!
 * procfs_mem_read() - Read process' memory.
 *
 *      @param  pmemfd  The file descriptor returned by procfs_mem_open() for
 *                      reading from the process' memory.
 *
 *      @param  addr    The address in the process' virtual memory to read.
 *
 *      @param  dest    Pointer to a buffer to read the memory contents into.
 *
 *      @param  len     The number of bytes to read.
 *
 *      @return the number of bytes read.
 */
size_t
procfs_mem_read(int pmemfd, vm_offset_t addr, void *dest, size_t len)
{
        ssize_t rv;

        assert(pmemfd >= 0);

        rv = pread(pmemfd, dest, len, addr);
        if (rv < 0)
                fatal(EX_OSERR, "read(procfs): %m");

        return rv;
}


/*!
 * procfs_get_procname() - Get the name of the process with the given pid.
```

```
 *
 *      @param  pid     The process identifier to get the name of.
 *
 *      @returns a newly-allocated string containing the name of the process
 *               or NULL if the name could not be determined.
 *
 *      It is the caller's responsibility to free the returned string when
 *      it is done with it.
 */
char *
procfs_get_procname(pid_t pid)
{
        char buffer[NAME_MAX + 45];
        regex_t re_postname;
        regmatch_t re_match;
        int re_error;
        ssize_t len;
        char *pos;
        int fd;

        /*
         * Only /proc/XXX/status has the original process name, unfortunately
         * it is difficult to parse correctly.  The /proc/XXX/cmdline file
         * seems to be ideal, except that it maybe be altered by the
         * process and hence may have non-sensical values (e.g. sendmail which
         * changes its name for status reporting).
         */
        fd = procfs_generic_open(pid, "status");
        if (fd < 0)
                return NULL;

        /*
         * The /proc/XXX/status file is only a single line.  Of that, we only
         * need to read the process name (maximum NAME_MAX chars) plus some
         * trailing text to identify where the process name ends.
         */
        len = read(fd, buffer, sizeof(buffer) - 1);
        if (len < 0) {
                procfs_generic_close(&fd);
                return NULL;
        }

        procfs_generic_close(&fd);

        /*
         * Now for the trick of parsing the status line.  The format is a
         * space-separated list of various fields.  The issue is how to
         * accurately identify the process name which itself may have spaces
         * embedded in it.  The solution: don't try to find the process name
         * but rather the text immediately following the process name.  Once
         * we have found that, we know everything before that is the process
         * name (spaces and all).
         */
        memset(&re_postname, 0, sizeof(re_postname));
        re_error = regcomp(&re_postname,
                           /* my cat 83162 82755 83162 82755 5,8 ctty ... */
                           /*        ^-------------------------^         */
                            "( [[:digit:]]{1,5}){4} [[:digit:]]+,[[:digit:]]+ ",
                            REG_EXTENDED);
        if (re_error) {
                regerror(re_error, &re_postname, buffer, sizeof(buffer));
                fatal(EX_SOFTWARE, "failed to compile regex: %s", buffer);
        }

        buffer[len] = '\0';
```

```
        re_error = regexec(&re_postname, buffer, 1, &re_match, 0);
        if (re_error) {
                regerror(re_error, &re_postname, buffer, sizeof(buffer));
                fatal(EX_SOFTWARE, "regex match failed: %s", buffer);
        }

        regfree(&re_postname);

        /*
         * Replace the first character matched with a nul-terminator.
         * Everything before that is the actual process name.
         */
        pos = buffer + re_match.rm_so;
        *pos = '\0';

        /*
         * Make a copy of the process name to return to the caller.  We don't
         * have to check for strdup() returning NULL because if it does it
         * just tells our caller we couldn't get the process name.
         */
        return strdup(buffer);
}
```

```
/*
 * Copyright (c) 2004 Kelly Yancey
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $kbyanc: dyntrace/dyntrace/ptrace.c,v 1.8 2004/12/27 04:31:54 kbyanc Exp $
 */

#include <sys/types.h>
#include <sys/ptrace.h>
#include <sys/wait.h>

#include <assert.h>
#include <ctype.h>
#include <errno.h>
#include <signal.h>
#include <stdbool.h>
#include <stdlib.h>
#include <sysexits.h>
#include <unistd.h>

#include <machine/reg.h>

#include "dyntrace.h"
#include "ptrace.h"


struct ptrace_state {
        enum { ATTACHED, DETACHED, TERMINATED } status;
        pid_t    pid;
        int      signum;
};

static bool      ptrace_initialized = false;

static const char *ptrace_signal_name(int sig);
static void      ptrace_sig_ignore(int sig);
static ptstate_t ptrace_alloc(pid_t pid);


/*!
 * ptrace_signal_name() - Map signal numbers to signal names.
 *
 *      @param  sig     The signal number to get the name of.
 *
```

```
 *      @return pointer to static storage holding the signal name string.
 *
 *      The caller must not try to free() the returned pointer.
 */
const char *
ptrace_signal_name(int sig)
{
        static char buffer[20];
        char *pos;

        buffer[sizeof(buffer) - 1] = '\0';

        if (sig >= 0 && sig < NSIG) {
                snprintf(buffer, sizeof(buffer) - 1, "sig%s", sys_signame[sig]);
                for (pos = buffer; *pos != '\0'; pos++)
                        *pos = toupper(*pos);
        } else
                snprintf(buffer, sizeof(buffer) - 1, "signal #%d", sig);

        return buffer;
}


/*!
 * ptrace_init() - Initialize ptrace interface API.
 *
 *      Initializes the local data structures used for interfacing with
 *      the ptrace API.  Installs a SIGCHLD signal handler.
 */
void
ptrace_init(void)
{

        /*
         * The traced process receives a SIGTRAP each time it stops under the
         * control of ptrace(2).  However, as the tracing process, we have
         * the opportunity to intercept the (fatal) signal if we have a
         * SIGCHLD handler other than the default SIG_IGN.  Since we wait
         * for the child to stop with waitpid(2), we install our own SIGCHLD
         * handler to ignore the signals.
         */
        setsighandler(SIGCHLD, ptrace_sig_ignore);

        ptrace_initialized = true;
}


/*!
 * ptrace_sig_ignore() - Stub signal handler for ignoring SIGCHLD signals.
 *
 *      Installing a stub no-op signal handler is different than using SIG_IGN
 *      as the action of SIGCHLD.  The former causes the child to stop or
 *      exit such that we can retreive the child's status with the wait(2)
 *      system call whereas the latter prevents us from learning the child's
 *      status altogether.
 */
void
ptrace_sig_ignore(int sig __unused)
{
}


/*!
 * ptrace_alloc() - Internal routine to allocate and initialize a ptrace
 *                  state handle.
```

```
 *
 *      @param  pid     The process identifer of the process to be traced.
 *
 *      @return newly-allocated ptrace state handle.
 */
ptstate_t
ptrace_alloc(pid_t pid)
{
        ptstate_t pts;

        pts = malloc(sizeof(*pts));
        if (pts == NULL)
                fatal(EX_OSERR, "malloc: %m");

        pts->status = DETACHED;
        pts->pid = pid;
        pts->signum = 0;

        return pts;
}


/*!
 * ptrace_fork() -
 *
 *      Wraps the fork(2) system call with additional logic for attaching to
 *      the child process for tracing.  As with fork(2), both the parent
 *      and the child process return; the calling code can distinguish which
 *      process it the child because it will return NULL whereas the parent
 *      will return a non-NULL ptrace handle.
 *
 *      @param  pidp    Pointer to pid_t to populate with the process
 *                      identifier of the child process.
 *
 *      @return NULL to the child process or ptrace state handle for tracing
 *              the child process to the parent process.
 */
ptstate_t
ptrace_fork(pid_t *pidp)
{
        ptstate_t pts;
        pid_t pid;

        if (!ptrace_initialized)
                ptrace_init();

        pid = fork();
        if (pid < 0)
                fatal(EX_OSERR, "fork: %m");
        if (pid == 0) {
                /*
                 * Child process.
                 * Set ourself up to be traced; a SIGTRAP will be raised on
                 * the first instruction after exec(3)'ing a new process image.
                 */
                if (ptrace(PT_TRACE_ME, 0, 0, 0) < 0)
                        fatal(EX_OSERR, "ptrace(PT_TRACE_ME): %m");

                return NULL;
        }

        /*
         * Parent process.
         * Wait for the child process to stop (specifically stopped due to
         * tracing as opposed to SIGSTOP), indicating it is ready to be traced.
```

```
         */
        pts = ptrace_alloc(pid);
        pts->status = ATTACHED;

        if (!ptrace_wait(pts))
                exit(EX_UNAVAILABLE);

        if (pidp != NULL)
                *pidp = pid;
        return pts;
}


/*!
 * ptrace_attach() - Attach to an existing process for tracing.
 *
 *      @param  pid     The process identifier to attach to.
 *
 *      @return ptrace handle for tracing the given process.
 *
 *      If the current process does not have sufficient permissions to trace
 *      the specified target process, an error is logged and the program will
 *      terminate.
 */
ptstate_t
ptrace_attach(pid_t pid)
{
        ptstate_t pts;

        if (!ptrace_initialized)
                ptrace_init();

        if (ptrace(PT_ATTACH, pid, 0, 0) < 0)
                fatal(EX_OSERR, "failed to attach to %u: %m", pid);

        pts = ptrace_alloc(pid);
        pts->status = ATTACHED;

        /* Wait for the traced process to stop. */
        if (!ptrace_wait(pts))
                exit(EX_UNAVAILABLE);

        return pts;
}


/*!
 * ptrace_detach() - Stop tracing a process, allowing it to continue running
 *                   as usual.
 *
 *      @param  pts     The ptrace handle for the process to stop tracing.
 *
 *      Detaching from a child process may cause it to terminate on some
 *      platforms.
 */
void
ptrace_detach(ptstate_t pts)
{

        assert(pts->status == ATTACHED);

        if (ptrace(PT_DETACH, pts->pid, (caddr_t)1, pts->signum) < 0)
                warn("failed to detach from %u: %m", pts->pid);
        pts->status = DETACHED;
        pts->signum = 0;
```

```
}


/*!
 * ptrace_done() - Free memory allocated to ptrace state handle.
 *
 *      @param  ptsp    Pointer to the ptrace state handle to free.
 *
 *      @post   The value is ptsp points to is invalidated so it cannot be
 *              passed to any ptrace_* routine.
 */
void
ptrace_done(ptstate_t *ptsp)
{
        ptstate_t pts = *ptsp;

        *ptsp = NULL;
        free(pts);
}


/*!
 * ptrace_step() - Single-step the given process by a single instruction.
 *
 *      Allows the process controlled by the given ptrace state handle to
 *      execute a single instruction before stopping.
 *
 *      @param  pts     The ptrace state handle for the process to single-step.
 *
 *      @post   The ptrace_wait() routine should be called to wait for the
 *              process to stop again after executing the instruction.
 */
void
ptrace_step(ptstate_t pts)
{

        assert(pts->status == ATTACHED);

        if (pts->signum != 0) {
                debug("sending %s to %u",
                      ptrace_signal_name(pts->signum), pts->pid);
        }

        if (ptrace(PT_STEP, pts->pid, (caddr_t)1, pts->signum) < 0)
                fatal(EX_OSERR, "ptrace(PT_STEP, %u): %m", pts->pid);
}


/*!
 * ptrace_continue() - Continue the given process' execution.
 *
 *      Allows the process controlled by the given ptrace state handle to
 *      continue execution.  Execution continues until the process receives
 *      a signal or encounters a breakpoint.
 *
 *      @param  pts     The ptrace state handle for the process to unstop.
 */
void
ptrace_continue(ptstate_t pts)
{

        assert(pts->status == ATTACHED);

        if (pts->signum != 0) {
                debug("sending %s to %u",
```

```
                        ptrace_signal_name(pts->signum), pts->pid);
        }

        if (ptrace(PT_CONTINUE, pts->pid, (caddr_t)1, pts->signum) < 0)
                fatal(EX_OSERR, "ptrace(PT_CONTINUE, %u): %m", pts->pid);
}


/*!
 * ptrace_wait() - Wait for a process to stop.
 *
 *      Waits for the process controlled by the given state handle to stop.
 *
 *      @param  pts     The ptrace state handle for the process to wait for.
 *
 *      @return boolean true if the process has stopped; boolean false if the
 *              the process has terminated.
 */
bool
ptrace_wait(ptstate_t pts)
{
        int status;

        while (waitpid(pts->pid, &status, 0) < 0) {
                if (errno != EINTR)
                        fatal(EX_OSERR, "waitpid(%u): %m", pts->pid);
        }

        /*
         * The normal case is that the process is stopped.  If the process
         * stopped due to a signal other than SIGTRAP then record that signal
         * so we can send it to the process when we continue its execution.
         * SIGTRAPs are generated due to our tracing of the process.
         */
        if (WIFSTOPPED(status)) {
                pts->signum = WSTOPSIG(status);
                if (pts->signum == SIGTRAP)
                        pts->signum = 0;
                return true;
        }

        if (WIFEXITED(status)) {
                warn("pid %u exited with status %u", pts->pid,
                     WEXITSTATUS(status));
                pts->status = TERMINATED;
                return false;
        }

        if (WIFSIGNALED(status)) {
                warn("pid %u exited on %s", pts->pid,
                     ptrace_signal_name(WTERMSIG(status)));
                pts->status = TERMINATED;
                return false;
        }

        assert(0);
        /* NOTREACHED */
        return true;
}


/*!
 * ptrace_signal() - Send a signal to a process.
 *
 *      @param  pts     The state handle of the process to signal.
```

```
 *
 *      @param  signum  The signal number to send to the process.
 *
 *      The specified signal is sent to the process when it resumes execution
 *      either by ptrace_step(), ptrace_continue(), or ptrace_detach().
 */
void
ptrace_signal(ptstate_t pts, int signum)
{

        assert(pts->status == ATTACHED);

        if (signum != SIGTRAP)
                pts->signum = signum;
}


/*!
 * ptrace_getregs() - Get the values in the CPU registers for a process.
 *
 *      @param  pts     The state handle of the process to read the register
 *                      values from.
 *
 *      @param  regs    Machine-dependent structure to populate with the
 *                      target process' register values.
 *
 *      @pre    The process controlled by the given state handle must be
 *              stopped.
 */
void
ptrace_getregs(ptstate_t pts, struct reg *regs)
{

        assert(pts->status == ATTACHED);

        if (ptrace(PT_GETREGS, pts->pid, (caddr_t)regs, 0) < 0)
                fatal(EX_OSERR, "ptrace(PT_GETREGS, %u): %m", pts->pid);
}


/*!
 * ptrace_setregs() - Set the values of the CPU registers for a process.
 *
 *      @param  pts     The state handle of the process to write the register
 *                      values for.
 *
 *      @param  regs    Machine-dependent structure to load the target process'
 *                      register values from.
 *
 *      @pre    The process controlled by the given state handle must be
 *              stopped.
 */
void
ptrace_setregs(ptstate_t pts, const struct reg *regs)
{

        assert(pts->status == ATTACHED);

        if (ptrace(PT_SETREGS, pts->pid, __DECONST(caddr_t, regs), 0) < 0)
                fatal(EX_OSERR, "ptrace(PT_GETREGS, %u): %m", pts->pid);
}


/*!
 * ptrace_read() - Read the contents of a process' virtual memory.
```

```
 *
 *      @param  pts     The state handle of the process to read from.
 *
 *      @param  addr    The address in the given process' virtual memory to
 *                      read.
 *
 *      @param  dest    Pointer to buffer in the current process to read the
 *                      memory contents into.
 *
 *      @param  len     The number of bytes to read.
 *
 *      @return the actual number of bytes read.
 */
size_t
ptrace_read(ptstate_t pts, vm_offset_t addr, void *dest, size_t len)
{
        struct ptrace_io_desc pio;

        assert(pts->status == ATTACHED);
        assert(sizeof(addr) >= sizeof(void *));

        pio.piod_op = PIOD_READ_I;
        pio.piod_offs = (void *)(uintptr_t)addr;
        pio.piod_addr = dest;
        pio.piod_len = len;

        if (ptrace(PT_IO, pts->pid, (caddr_t)&pio, 0) < 0) {
                fatal(EX_OSERR, "ptrace(PT_IO, %u, 0x%08x, %u): %m",
                      pts->pid, addr, len);
        }

        return pio.piod_len;
}


/*!
 * ptrace_write() - Write the contents of a process' virtual memory.
 *
 *      @param  pts     The state handle of the process to write to.
 *
 *      @param  addr    The address in the process' virtual memory to write to.
 *
 *      @param  src     Pointer to buffer in the current process containing
 *                      the data to write.
 *
 *      @param  len     The number of bytes to write.
 */
void
ptrace_write(ptstate_t pts, vm_offset_t addr, const void *src, size_t len)
{
        struct ptrace_io_desc pio;

        assert(pts->status == ATTACHED);
        assert(sizeof(addr) >= sizeof(void *));

        while (len > 0) {
                pio.piod_op = PIOD_WRITE_I;
                pio.piod_offs = (void *)(uintptr_t)addr;
                pio.piod_addr = __DECONST(void *, src);
                pio.piod_len = len;

                if (ptrace(PT_IO, pts->pid, (caddr_t)&pio, 0) < 0) {
                        fatal(EX_OSERR, "ptrace(PT_IO, %u, 0x%08x, %u): %m",
                              pts->pid, addr, len);
                }
```

```
                src = ((const uint8_t *)src) + pio.piod_len;
                addr += pio.piod_len;
                len -= pio.piod_len;
        }
}
```

```
/*
 * Copyright (c) 2004 Kelly Yancey
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $kbyanc: dyntrace/dyntrace/ptrace.h,v 1.3 2004/12/23 01:45:19 kbyanc Exp $
 */

#ifndef _INCLUDE_DYNTRACE_PTRACE_H
#define _INCLUDE_DYNTRACE_PTRACE_H

#include <sys/cdefs.h>
#include <stdbool.h>

struct reg;     /* Defined in <machine/reg.h> */

typedef struct ptrace_state *ptstate_t;


__BEGIN_DECLS

extern void      ptrace_init(void);
extern ptstate_t ptrace_fork(pid_t *pidp);
extern ptstate_t ptrace_attach(pid_t pid);
extern void      ptrace_detach(ptstate_t pts);
extern void      ptrace_done(ptstate_t *ptsp);
extern void      ptrace_step(ptstate_t pts);
extern void      ptrace_continue(ptstate_t pts);
extern bool      ptrace_wait(ptstate_t pts);
extern void      ptrace_signal(ptstate_t pts, int signum);
extern void      ptrace_getregs(ptstate_t pts, struct reg *regs);
extern void      ptrace_setregs(ptstate_t pts, const struct reg *regs);
extern size_t    ptrace_read(ptstate_t pts, vm_offset_t addr,
                             void *dest, size_t len);
extern void      ptrace_write(ptstate_t pts, vm_offset_t addr,
                              const void *src, size_t len);

__END_DECLS

#endif
```

```
/*
 * Copyright (c) 1988, 1989, 1993
 *      The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 4. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *      @(#)radix.c     8.5 (Berkeley) 5/19/95
 * $FreeBSD: src/sys/net/radix.c,v 1.36 2004/04/21 15:27:36 luigi Exp $
 * $kbyanc: dyntrace/dyntrace/radix.c,v 1.4 2004/12/23 01:45:19 kbyanc Exp $
 */

/*
 * Routines to build and maintain radix trees for routing lookups.
 */

#include <sys/types.h>
#include <sys/syslog.h>

#include <stdlib.h>
#include <string.h>
#include <sysexits.h>

#include "dyntrace.h"
#include "radix.h"

#define log(x, ...)     warn(__VA_ARGS__)
#define panic(s)        fatal(EX_SOFTWARE, "%s", s);
#define min(a,b)        (((a)<(b))?(a):(b))

static int      rn_walktree_from(struct radix_node_head *h, void *a, void *m,
                    walktree_f_t *f, void *w);
static int rn_walktree(struct radix_node_head *, walktree_f_t *, void *);
static struct radix_node
         *rn_insert(void *, struct radix_node_head *, int *,
             struct radix_node [2]),
         *rn_newpair(void *, int, struct radix_node[2]),
         *rn_search(void *, struct radix_node *),
         *rn_search_m(void *, struct radix_node *, void *);

int     max_keylen = 0;
static struct radix_mask *rn_mkfreelist;
static struct radix_node_head *mask_rnhead;
/*
```

```
 * Work area -- the following point to 3 buffers of size max_keylen,
 * allocated in this order in a block of memory malloc'ed by rn_init.
 */
static char *rn_zeros, *rn_ones, *addmask_key;

#define MKGet(m) {                                              \
        if (rn_mkfreelist) {                                    \
                m = rn_mkfreelist;                              \
                rn_mkfreelist = (m)->rm_mklist;                 \
        } else                                                  \
                R_Malloc(m, struct radix_mask *, sizeof (struct radix_mask)); }

#define MKFree(m) { (m)->rm_mklist = rn_mkfreelist; rn_mkfreelist = (m);}

#define rn_masktop (mask_rnhead->rnh_treetop)

static int      rn_lexobetter(void *m_arg, void *n_arg);
static struct radix_mask *
                rn_new_radix_mask(struct radix_node *tt,
                    struct radix_mask *next);
static int      rn_satisfies_leaf(char *trial, struct radix_node *leaf,
                    int skip);

/*
 * The data structure for the keys is a radix tree with one way
 * branching removed.  The index rn_bit at an internal node n represents a bit
 * position to be tested.  The tree is arranged so that all descendants
 * of a node n have keys whose bits all agree up to position rn_bit - 1.
 * (We say the index of n is rn_bit.)
 *
 * There is at least one descendant which has a one bit at position rn_bit,
 * and at least one with a zero there.
 *
 * A route is determined by a pair of key and mask.  We require that the
 * bit-wise logical and of the key and mask to be the key.
 * We define the index of a route to associated with the mask to be
 * the first bit number in the mask where 0 occurs (with bit number 0
 * representing the highest order bit).
 *
 * We say a mask is normal if every bit is 0, past the index of the mask.
 * If a node n has a descendant (k, m) with index(m) == index(n) == rn_bit,
 * and m is a normal mask, then the route applies to every descendant of n.
 * If the index(m) < rn_bit, this implies the trailing last few bits of k
 * before bit b are all 0, (and hence consequently true of every descendant
 * of n), so the route applies to all descendants of the node as well.
 *
 * Similar logic shows that a non-normal mask m such that
 * index(m) <= index(n) could potentially apply to many children of n.
 * Thus, for each non-host route, we attach its mask to a list at an internal
 * node as high in the tree as we can go.
 *
 * The present version of the code makes use of normal routes in short-
 * circuiting an explict mask and compare operation when testing whether
 * a key satisfies a normal route, and also in remembering the unique leaf
 * that governs a subtree.
 */

/*
 * Most of the functions in this code assume that the key/mask arguments
 * are sockaddr-like structures, where the first byte is an u_char
 * indicating the size of the entire structure.
 *
 * To make the assumption more explicit, we use the LEN() macro to access
 * this field. It is safe to pass an expression with side effects
 * to LEN() as the argument is evaluated only once.
```

```
 */
#define LEN(x) (*(const u_char *)(x))

/*
 * XXX THIS NEEDS TO BE FIXED
 * In the code, pointers to keys and masks are passed as either
 * 'void *' (because callers use to pass pointers of various kinds), or
 * 'caddr_t' (which is fine for pointer arithmetics, but not very
 * clean when you dereference it to access data). Furthermore, caddr_t
 * is really 'char *', while the natural type to operate on keys and
 * masks would be 'u_char'. This mismatch require a lot of casts and
 * intermediate variables to adapt types that clutter the code.
 */

/*
 * Search a node in the tree matching the key.
 */
static struct radix_node *
rn_search(v_arg, head)
        void *v_arg;
        struct radix_node *head;
{
        register struct radix_node *x;
        register caddr_t v;

        for (x = head, v = v_arg; x->rn_bit >= 0;) {
                if (x->rn_bmask & v[x->rn_offset])
                        x = x->rn_right;
                else
                        x = x->rn_left;
        }
        return (x);
}

/*
 * Same as above, but with an additional mask.
 * XXX note this function is used only once.
 */
static struct radix_node *
rn_search_m(v_arg, head, m_arg)
        struct radix_node *head;
        void *v_arg, *m_arg;
{
        register struct radix_node *x;
        register caddr_t v = v_arg, m = m_arg;

        for (x = head; x->rn_bit >= 0;) {
                if ((x->rn_bmask & m[x->rn_offset]) &&
                    (x->rn_bmask & v[x->rn_offset]))
                        x = x->rn_right;
                else
                        x = x->rn_left;
        }
        return x;
}

int
rn_refines(m_arg, n_arg)
        void *m_arg, *n_arg;
{
        register caddr_t m = m_arg, n = n_arg;
        register caddr_t lim, lim2 = lim = n + LEN(n);
        int longer = LEN(n++) - (int)LEN(m++);
        int masks_are_equal = 1;
```

```
        if (longer > 0)
                lim -= longer;
        while (n < lim) {
                if (*n & ~(*m))
                        return 0;
                if (*n++ != *m++)
                        masks_are_equal = 0;
        }
        while (n < lim2)
                if (*n++)
                        return 0;
        if (masks_are_equal && (longer < 0))
                for (lim2 = m - longer; m < lim2; )
                        if (*m++)
                                return 1;
        return (!masks_are_equal);
}

struct radix_node *
rn_lookup(v_arg, m_arg, head)
        void *v_arg, *m_arg;
        struct radix_node_head *head;
{
        register struct radix_node *x;
        caddr_t netmask = 0;

        if (m_arg) {
                x = rn_addmask(m_arg, 1, head->rnh_treetop->rn_offset);
                if (x == 0)
                        return (0);
                netmask = x->rn_key;
        }
        x = rn_match(v_arg, head);
        if (x && netmask) {
                while (x && x->rn_mask != netmask)
                        x = x->rn_dupedkey;
        }
        return x;
}

static int
rn_satisfies_leaf(trial, leaf, skip)
        char *trial;
        register struct radix_node *leaf;
        int skip;
{
        register char *cp = trial, *cp2 = leaf->rn_key, *cp3 = leaf->rn_mask;
        char *cplim;
        int length = min(LEN(cp), LEN(cp2));

        if (cp3 == 0)
                cp3 = rn_ones;
        else
                length = min(length, *(u_char *)cp3);
        cplim = cp + length; cp3 += skip; cp2 += skip;
        for (cp += skip; cp < cplim; cp++, cp2++, cp3++)
                if ((*cp ^ *cp2) & *cp3)
                        return 0;
        return 1;
}

struct radix_node *
rn_match(v_arg, head)
        void *v_arg;
        struct radix_node_head *head;
```

```
{
        caddr_t v = v_arg;
        register struct radix_node *t = head->rnh_treetop, *x;
        register caddr_t cp = v, cp2;
        caddr_t cplim;
        struct radix_node *saved_t, *top = t;
        int off = t->rn_offset, vlen = LEN(cp), matched_off;
        register int test, b, rn_bit;

        /*
         * Open code rn_search(v, top) to avoid overhead of extra
         * subroutine call.
         */
        for (; t->rn_bit >= 0; ) {
                if (t->rn_bmask & cp[t->rn_offset])
                        t = t->rn_right;
                else
                        t = t->rn_left;
        }
        /*
         * See if we match exactly as a host destination
         * or at least learn how many bits match, for normal mask finesse.
         *
         * It doesn't hurt us to limit how many bytes to check
         * to the length of the mask, since if it matches we had a genuine
         * match and the leaf we have is the most specific one anyway;
         * if it didn't match with a shorter length it would fail
         * with a long one.  This wins big for class B&C netmasks which
         * are probably the most common case...
         */
        if (t->rn_mask)
                vlen = *(u_char *)t->rn_mask;
        cp += off; cp2 = t->rn_key + off; cplim = v + vlen;
        for (; cp < cplim; cp++, cp2++)
                if (*cp != *cp2)
                        goto on1;
        /*
         * This extra grot is in case we are explicitly asked
         * to look up the default.  Ugh!
         *
         * Never return the root node itself, it seems to cause a
         * lot of confusion.
         */
        if (t->rn_flags & RNF_ROOT)
                t = t->rn_dupedkey;
        return t;
on1:
        test = (*cp ^ *cp2) & 0xff; /* find first bit that differs */
        for (b = 7; (test >>= 1) > 0;)
                b--;
        matched_off = cp - v;
        b += matched_off << 3;
        rn_bit = -1 - b;
        /*
         * If there is a host route in a duped-key chain, it will be first.
         */
        if ((saved_t = t)->rn_mask == 0)
                t = t->rn_dupedkey;
        for (; t; t = t->rn_dupedkey)
                /*
                 * Even if we don't match exactly as a host,
                 * we may match if the leaf we wound up at is
                 * a route to a net.
                 */
                if (t->rn_flags & RNF_NORMAL) {
```

```
                        if (rn_bit <= t->rn_bit)
                                return t;
                } else if (rn_satisfies_leaf(v, t, matched_off))
                                return t;
        t = saved_t;
        /* start searching up the tree */
        do {
                register struct radix_mask *m;
                t = t->rn_parent;
                m = t->rn_mklist;
                /*
                 * If non-contiguous masks ever become important
                 * we can restore the masking and open coding of
                 * the search and satisfaction test and put the
                 * calculation of "off" back before the "do".
                 */
                while (m) {
                        if (m->rm_flags & RNF_NORMAL) {
                                if (rn_bit <= m->rm_bit)
                                        return (m->rm_leaf);
                        } else {
                                off = min(t->rn_offset, matched_off);
                                x = rn_search_m(v, t, m->rm_mask);
                                while (x && x->rn_mask != m->rm_mask)
                                        x = x->rn_dupedkey;
                                if (x && rn_satisfies_leaf(v, x, off))
                                        return x;
                        }
                        m = m->rm_mklist;
                }
        } while (t != top);
        return 0;
}

#ifdef RN_DEBUG
int     rn_nodenum;
struct  radix_node *rn_clist;
int     rn_saveinfo;
int     rn_debug =  1;
#endif

/*
 * Whenever we add a new leaf to the tree, we also add a parent node,
 * so we allocate them as an array of two elements: the first one must be
 * the leaf (see RNTORT() in route.c), the second one is the parent.
 * This routine initializes the relevant fields of the nodes, so that
 * the leaf is the left child of the parent node, and both nodes have
 * (almost) all all fields filled as appropriate.
 * (XXX some fields are left unset, see the '#if 0' section).
 * The function returns a pointer to the parent node.
 */

static struct radix_node *
rn_newpair(v, b, nodes)
        void *v;
        int b;
        struct radix_node nodes[2];
{
        register struct radix_node *tt = nodes, *t = tt + 1;
        t->rn_bit = b;
        t->rn_bmask = 0x80 >> (b & 7);
        t->rn_left = tt;
        t->rn_offset = b >> 3;

#if 0  /* XXX perhaps we should fill these fields as well. */
```

```
        t->rn_parent = t->rn_right = NULL;

        tt->rn_mask = NULL;
        tt->rn_dupedkey = NULL;
        tt->rn_bmask = 0;
#endif
        tt->rn_bit = -1;
        tt->rn_key = (caddr_t)v;
        tt->rn_parent = t;
        tt->rn_flags = t->rn_flags = RNF_ACTIVE;
        tt->rn_mklist = t->rn_mklist = 0;
#ifdef RN_DEBUG
        tt->rn_info = rn_nodenum++; t->rn_info = rn_nodenum++;
        tt->rn_twin = t;
        tt->rn_ybro = rn_clist;
        rn_clist = tt;
#endif
        return t;
}

static struct radix_node *
rn_insert(v_arg, head, dupentry, nodes)
        void *v_arg;
        struct radix_node_head *head;
        int *dupentry;
        struct radix_node nodes[2];
{
        caddr_t v = v_arg;
        struct radix_node *top = head->rnh_treetop;
        int head_off = top->rn_offset, vlen = (int)LEN(v);
        register struct radix_node *t = rn_search(v_arg, top);
        register caddr_t cp = v + head_off;
        register int b;
        struct radix_node *tt;
        /*
         * Find first bit at which v and t->rn_key differ
         */
    {
        register caddr_t cp2 = t->rn_key + head_off;
        register int cmp_res;
        caddr_t cplim = v + vlen;

        while (cp < cplim)
                if (*cp2++ != *cp++)
                        goto on1;
        *dupentry = 1;
        return t;
on1:
        *dupentry = 0;
        cmp_res = (cp[-1] ^ cp2[-1]) & 0xff;
        for (b = (cp - v) << 3; cmp_res; b--)
                cmp_res >>= 1;
    }
    {
        register struct radix_node *p, *x = top;
        cp = v;
        do {
                p = x;
                if (cp[x->rn_offset] & x->rn_bmask)
                        x = x->rn_right;
                else
                        x = x->rn_left;
        } while ((unsigned)b > (unsigned)x->rn_bit);
                                /* x->rn_bit < b && x->rn_bit >= 0 */
#ifdef RN_DEBUG
```

```
        if (rn_debug)
                log(LOG_DEBUG, "rn_insert: Going In:\n"), traverse(p);
#endif
        t = rn_newpair(v_arg, b, nodes);
        tt = t->rn_left;
        if ((cp[p->rn_offset] & p->rn_bmask) == 0)
                p->rn_left = t;
        else
                p->rn_right = t;
        x->rn_parent = t;
        t->rn_parent = p; /* frees x, p as temp vars below */
        if ((cp[t->rn_offset] & t->rn_bmask) == 0) {
                t->rn_right = x;
        } else {
                t->rn_right = tt;
                t->rn_left = x;
        }
#ifdef RN_DEBUG
        if (rn_debug)
                log(LOG_DEBUG, "rn_insert: Coming Out:\n"), traverse(p);
#endif
    }
        return (tt);
}

struct radix_node *
rn_addmask(n_arg, search, skip)
        int search, skip;
        void *n_arg;
{
        caddr_t netmask = (caddr_t)n_arg;
        register struct radix_node *x;
        register caddr_t cp, cplim;
        register int b = 0, mlen, j;
        int maskduplicated, m0, isnormal;
        struct radix_node *saved_x;
        static int last_zeroed = 0;

        if ((mlen = LEN(netmask)) > max_keylen)
                mlen = max_keylen;
        if (skip == 0)
                skip = 1;
        if (mlen <= skip)
                return (mask_rnhead->rnh_nodes);
        if (skip > 1)
                bcopy(rn_ones + 1, addmask_key + 1, skip - 1);
        if ((m0 = mlen) > skip)
                bcopy(netmask + skip, addmask_key + skip, mlen - skip);
        /*
         * Trim trailing zeroes.
         */
        for (cp = addmask_key + mlen; (cp > addmask_key) && cp[-1] == 0;)
                cp--;
        mlen = cp - addmask_key;
        if (mlen <= skip) {
                if (m0 >= last_zeroed)
                        last_zeroed = mlen;
                return (mask_rnhead->rnh_nodes);
        }
        if (m0 < last_zeroed)
                bzero(addmask_key + m0, last_zeroed - m0);
        *addmask_key = last_zeroed = mlen;
        x = rn_search(addmask_key, rn_masktop);
        if (bcmp(addmask_key, x->rn_key, mlen) != 0)
                x = 0;
```

```
        if (x || search)
                return (x);
        R_Zalloc(x, struct radix_node *, max_keylen + 2 * sizeof (*x));
        if ((saved_x = x) == 0)
                return (0);
        netmask = cp = (caddr_t)(x + 2);
        bcopy(addmask_key, cp, mlen);
        x = rn_insert(cp, mask_rnhead, &maskduplicated, x);
        if (maskduplicated) {
                log(LOG_ERR, "rn_addmask: mask impossibly already in tree");
                Free(saved_x);
                return (x);
        }
        /*
         * Calculate index of mask, and check for normalcy.
         * First find the first byte with a 0 bit, then if there are
         * more bits left (remember we already trimmed the trailing 0's),
         * the pattern must be one of those in normal_chars[], or we have
         * a non-contiguous mask.
         */
        cplim = netmask + mlen;
        isnormal = 1;
        for (cp = netmask + skip; (cp < cplim) && *(u_char *)cp == 0xff;)
                cp++;
        if (cp != cplim) {
                static char normal_chars[] = {
                        0, 0x80, 0xc0, 0xe0, 0xf0, 0xf8, 0xfc, 0xfe, 0xff};

                for (j = 0x80; (j & *cp) != 0; j >>= 1)
                        b++;
                if (*cp != normal_chars[b] || cp != (cplim - 1))
                        isnormal = 0;
        }
        b += (cp - netmask) << 3;
        x->rn_bit = -1 - b;
        if (isnormal)
                x->rn_flags |= RNF_NORMAL;
        return (x);
}

static int      /* XXX: arbitrary ordering for non-contiguous masks */
rn_lexobetter(m_arg, n_arg)
        void *m_arg, *n_arg;
{
        register u_char *mp = m_arg, *np = n_arg, *lim;

        if (LEN(mp) > LEN(np))
                return 1;  /* not really, but need to check longer one first */
        if (LEN(mp) == LEN(np))
                for (lim = mp + LEN(mp); mp < lim;)
                        if (*mp++ > *np++)
                                return 1;
        return 0;
}

static struct radix_mask *
rn_new_radix_mask(tt, next)
        register struct radix_node *tt;
        register struct radix_mask *next;
{
        register struct radix_mask *m;

        MKGet(m);
        if (m == 0) {
                log(LOG_ERR, "Mask for route not entered\n");
```

```
                return (0);
        }
        bzero(m, sizeof *m);
        m->rm_bit = tt->rn_bit;
        m->rm_flags = tt->rn_flags;
        if (tt->rn_flags & RNF_NORMAL)
                m->rm_leaf = tt;
        else
                m->rm_mask = tt->rn_mask;
        m->rm_mklist = next;
        tt->rn_mklist = m;
        return m;
}

struct radix_node *
rn_addroute(v_arg, n_arg, head, treenodes)
        void *v_arg, *n_arg;
        struct radix_node_head *head;
        struct radix_node treenodes[2];
{
        caddr_t v = (caddr_t)v_arg, netmask = (caddr_t)n_arg;
        register struct radix_node *t, *x = 0, *tt;
        struct radix_node *saved_tt, *top = head->rnh_treetop;
        short b = 0, b_leaf = 0;
        int keyduplicated;
        caddr_t mmask;
        struct radix_mask *m, **mp;

        /*
         * In dealing with non-contiguous masks, there may be
         * many different routes which have the same mask.
         * We will find it useful to have a unique pointer to
         * the mask to speed avoiding duplicate references at
         * nodes and possibly save time in calculating indices.
         */
        if (netmask)  {
                if ((x = rn_addmask(netmask, 0, top->rn_offset)) == 0)
                        return (0);
                b_leaf = x->rn_bit;
                b = -1 - x->rn_bit;
                netmask = x->rn_key;
        }
        /*
         * Deal with duplicated keys: attach node to previous instance
         */
        saved_tt = tt = rn_insert(v, head, &keyduplicated, treenodes);
        if (keyduplicated) {
                for (t = tt; tt; t = tt, tt = tt->rn_dupedkey) {
                        if (tt->rn_mask == netmask)
                                return (0);
                        if (netmask == 0 ||
                            (tt->rn_mask &&
                             ((b_leaf < tt->rn_bit) /* index(netmask) > node */
                              || rn_refines(netmask, tt->rn_mask)
                              || rn_lexobetter(netmask, tt->rn_mask))))
                                break;
                }
                /*
                 * If the mask is not duplicated, we wouldn't
                 * find it among possible duplicate key entries
                 * anyway, so the above test doesn't hurt.
                 *
                 * We sort the masks for a duplicated key the same way as
                 * in a masklist -- most specific to least specific.
                 * This may require the unfortunate nuisance of relocating
```

```
                 * the head of the list.
                 *
                 * We also reverse, or doubly link the list through the
                 * parent pointer.
                 */
                if (tt == saved_tt) {
                        struct  radix_node *xx = x;
                        /* link in at head of list */
                        (tt = treenodes)->rn_dupedkey = t;
                        tt->rn_flags = t->rn_flags;
                        tt->rn_parent = x = t->rn_parent;
                        t->rn_parent = tt;                      /* parent */
                        if (x->rn_left == t)
                                x->rn_left = tt;
                        else
                                x->rn_right = tt;
                        saved_tt = tt; x = xx;
                } else {
                        (tt = treenodes)->rn_dupedkey = t->rn_dupedkey;
                        t->rn_dupedkey = tt;
                        tt->rn_parent = t;                      /* parent */
                        if (tt->rn_dupedkey)                    /* parent */
                                tt->rn_dupedkey->rn_parent = tt; /* parent */
                }
#ifdef RN_DEBUG
                t=tt+1; tt->rn_info = rn_nodenum++; t->rn_info = rn_nodenum++;
                tt->rn_twin = t; tt->rn_ybro = rn_clist; rn_clist = tt;
#endif
                tt->rn_key = (caddr_t) v;
                tt->rn_bit = -1;
                tt->rn_flags = RNF_ACTIVE;
        }
        /*
         * Put mask in tree.
         */
        if (netmask) {
                tt->rn_mask = netmask;
                tt->rn_bit = x->rn_bit;
                tt->rn_flags |= x->rn_flags & RNF_NORMAL;
        }
        t = saved_tt->rn_parent;
        if (keyduplicated)
                goto on2;
        b_leaf = -1 - t->rn_bit;
        if (t->rn_right == saved_tt)
                x = t->rn_left;
        else
                x = t->rn_right;
        /* Promote general routes from below */
        if (x->rn_bit < 0) {
            for (mp = &t->rn_mklist; x; x = x->rn_dupedkey)
                if (x->rn_mask && (x->rn_bit >= b_leaf) && x->rn_mklist == 0) {
                        *mp = m = rn_new_radix_mask(x, 0);
                        if (m)
                                mp = &m->rm_mklist;
                }
        } else if (x->rn_mklist) {
                /*
                 * Skip over masks whose index is > that of new node
                 */
                for (mp = &x->rn_mklist; (m = *mp); mp = &m->rm_mklist)
                        if (m->rm_bit >= b_leaf)
                                break;
                t->rn_mklist = m; *mp = 0;
        }
```

```
on2:
        /* Add new route to highest possible ancestor's list */
        if ((netmask == 0) || (b > t->rn_bit ))
                return tt; /* can't lift at all */
        b_leaf = tt->rn_bit;
        do {
                x = t;
                t = t->rn_parent;
        } while (b <= t->rn_bit && x != top);
        /*
         * Search through routes associated with node to
         * insert new route according to index.
         * Need same criteria as when sorting dupedkeys to avoid
         * double loop on deletion.
         */
        for (mp = &x->rn_mklist; (m = *mp); mp = &m->rm_mklist) {
                if (m->rm_bit < b_leaf)
                        continue;
                if (m->rm_bit > b_leaf)
                        break;
                if (m->rm_flags & RNF_NORMAL) {
                        mmask = m->rm_leaf->rn_mask;
                        if (tt->rn_flags & RNF_NORMAL) {
                            log(LOG_ERR,
                                "Non-unique normal route, mask not entered\n");
                                return tt;
                        }
                } else
                        mmask = m->rm_mask;
                if (mmask == netmask) {
                        m->rm_refs++;
                        tt->rn_mklist = m;
                        return tt;
                }
                if (rn_refines(netmask, mmask)
                    || rn_lexobetter(netmask, mmask))
                        break;
        }
        *mp = rn_new_radix_mask(tt, *mp);
        return tt;
}

struct radix_node *
rn_delete(v_arg, netmask_arg, head)
        void *v_arg, *netmask_arg;
        struct radix_node_head *head;
{
        register struct radix_node *t, *p, *x, *tt;
        struct radix_mask *m, *saved_m, **mp;
        struct radix_node *dupedkey, *saved_tt, *top;
        caddr_t v, netmask;
        int b, head_off, vlen;

        v = v_arg;
        netmask = netmask_arg;
        x = head->rnh_treetop;
        tt = rn_search(v, x);
        head_off = x->rn_offset;
        vlen =  LEN(v);
        saved_tt = tt;
        top = x;
        if (tt == 0 ||
            bcmp(v + head_off, tt->rn_key + head_off, vlen - head_off))
                return (0);
        /*
```

```
         * Delete our route from mask lists.
         */
        if (netmask) {
                if ((x = rn_addmask(netmask, 1, head_off)) == 0)
                        return (0);
                netmask = x->rn_key;
                while (tt->rn_mask != netmask)
                        if ((tt = tt->rn_dupedkey) == 0)
                                return (0);
        }
        if (tt->rn_mask == 0 || (saved_m = m = tt->rn_mklist) == 0)
                goto on1;
        if (tt->rn_flags & RNF_NORMAL) {
                if (m->rm_leaf != tt || m->rm_refs > 0) {
                        log(LOG_ERR, "rn_delete: inconsistent annotation\n");
                        return 0;  /* dangling ref could cause disaster */
                }
        } else {
                if (m->rm_mask != tt->rn_mask) {
                        log(LOG_ERR, "rn_delete: inconsistent annotation\n");
                        goto on1;
                }
                if (--m->rm_refs >= 0)
                        goto on1;
        }
        b = -1 - tt->rn_bit;
        t = saved_tt->rn_parent;
        if (b > t->rn_bit)
                goto on1; /* Wasn't lifted at all */
        do {
                x = t;
                t = t->rn_parent;
        } while (b <= t->rn_bit && x != top);
        for (mp = &x->rn_mklist; (m = *mp); mp = &m->rm_mklist)
                if (m == saved_m) {
                        *mp = m->rm_mklist;
                        MKFree(m);
                        break;
                }
        if (m == 0) {
                log(LOG_ERR, "rn_delete: couldn't find our annotation\n");
                if (tt->rn_flags & RNF_NORMAL)
                        return (0); /* Dangling ref to us */
        }
on1:
        /*
         * Eliminate us from tree
         */
        if (tt->rn_flags & RNF_ROOT)
                return (0);
#ifdef RN_DEBUG
        /* Get us out of the creation list */
        for (t = rn_clist; t && t->rn_ybro != tt; t = t->rn_ybro) {}
        if (t) t->rn_ybro = tt->rn_ybro;
#endif
        t = tt->rn_parent;
        dupedkey = saved_tt->rn_dupedkey;
        if (dupedkey) {
                /*
                 * Here, tt is the deletion target and
                 * saved_tt is the head of the dupekey chain.
                 */
                if (tt == saved_tt) {
                        /* remove from head of chain */
                        x = dupedkey; x->rn_parent = t;
```

```
                        if (t->rn_left == tt)
                                t->rn_left = x;
                        else
                                t->rn_right = x;
                } else {
                        /* find node in front of tt on the chain */
                        for (x = p = saved_tt; p && p->rn_dupedkey != tt;)
                                p = p->rn_dupedkey;
                        if (p) {
                                p->rn_dupedkey = tt->rn_dupedkey;
                                if (tt->rn_dupedkey)            /* parent */
                                        tt->rn_dupedkey->rn_parent = p;
                                                                /* parent */
                        } else log(LOG_ERR, "rn_delete: couldn't find us\n");
                }
                t = tt + 1;
                if  (t->rn_flags & RNF_ACTIVE) {
#ifndef RN_DEBUG
                        *++x = *t;
                        p = t->rn_parent;
#else
                        b = t->rn_info;
                        *++x = *t;
                        t->rn_info = b;
                        p = t->rn_parent;
#endif
                        if (p->rn_left == t)
                                p->rn_left = x;
                        else
                                p->rn_right = x;
                        x->rn_left->rn_parent = x;
                        x->rn_right->rn_parent = x;
                }
                goto out;
        }
        if (t->rn_left == tt)
                x = t->rn_right;
        else
                x = t->rn_left;
        p = t->rn_parent;
        if (p->rn_right == t)
                p->rn_right = x;
        else
                p->rn_left = x;
        x->rn_parent = p;
        /*
         * Demote routes attached to us.
         */
        if (t->rn_mklist) {
                if (x->rn_bit >= 0) {
                        for (mp = &x->rn_mklist; (m = *mp);)
                                mp = &m->rm_mklist;
                        *mp = t->rn_mklist;
                } else {
                        /* If there are any key,mask pairs in a sibling
                           duped-key chain, some subset will appear sorted
                           in the same order attached to our mklist */
                        for (m = t->rn_mklist; m && x; x = x->rn_dupedkey)
                                if (m == x->rn_mklist) {
                                        struct radix_mask *mm = m->rm_mklist;
                                        x->rn_mklist = 0;
                                        if (--(m->rm_refs) < 0)
                                                MKFree(m);
                                        m = mm;
                                }
```

```
                        if (m)
                                log(LOG_ERR,
                                    "rn_delete: Orphaned Mask %p at %p\n",
                                    (void *)m, (void *)x);
                }
        }
        /*
         * We may be holding an active internal node in the tree.
         */
        x = tt + 1;
        if (t != x) {
#ifndef RN_DEBUG
                *t = *x;
#else
                b = t->rn_info;
                *t = *x;
                t->rn_info = b;
#endif
                t->rn_left->rn_parent = t;
                t->rn_right->rn_parent = t;
                p = x->rn_parent;
                if (p->rn_left == x)
                        p->rn_left = t;
                else
                        p->rn_right = t;
        }
out:
        tt->rn_flags &= ~RNF_ACTIVE;
        tt[1].rn_flags &= ~RNF_ACTIVE;
        return (tt);
}

/*
 * This is the same as rn_walktree() except for the parameters and the
 * exit.
 */
static int
rn_walktree_from(h, a, m, f, w)
        struct radix_node_head *h;
        void *a, *m;
        walktree_f_t *f;
        void *w;
{
        int error;
        struct radix_node *base, *next;
        u_char *xa = (u_char *)a;
        u_char *xm = (u_char *)m;
        register struct radix_node *rn, *last = 0 /* shut up gcc */;
        int stopping = 0;
        int lastb;

        /*
         * rn_search_m is sort-of-open-coded here. We cannot use the
         * function because we need to keep track of the last node seen.
         */
        /* printf("about to search\n"); */
        for (rn = h->rnh_treetop; rn->rn_bit >= 0; ) {
                last = rn;
                /* printf("rn_bit %d, rn_bmask %x, xm[rn_offset] %x\n",
                       rn->rn_bit, rn->rn_bmask, xm[rn->rn_offset]); */
                if (!(rn->rn_bmask & xm[rn->rn_offset])) {
                        break;
                }
                if (rn->rn_bmask & xa[rn->rn_offset]) {
                        rn = rn->rn_right;
```

```
                } else {
                        rn = rn->rn_left;
                }
        }
        /* printf("done searching\n"); */

        /*
         * Two cases: either we stepped off the end of our mask,
         * in which case last == rn, or we reached a leaf, in which
         * case we want to start from the last node we looked at.
         * Either way, last is the node we want to start from.
         */
        rn = last;
        lastb = rn->rn_bit;

        /* printf("rn %p, lastb %d\n", rn, lastb);*/

        /*
         * This gets complicated because we may delete the node
         * while applying the function f to it, so we need to calculate
         * the successor node in advance.
         */
        while (rn->rn_bit >= 0)
                rn = rn->rn_left;

        while (!stopping) {
                /* printf("node %p (%d)\n", rn, rn->rn_bit); */
                base = rn;
                /* If at right child go back up, otherwise, go right */
                while (rn->rn_parent->rn_right == rn
                       && !(rn->rn_flags & RNF_ROOT)) {
                        rn = rn->rn_parent;

                        /* if went up beyond last, stop */
                        if (rn->rn_bit < lastb) {
                                stopping = 1;
                                /* printf("up too far\n"); */
                                /*
                                 * XXX we should jump to the 'Process leaves'
                                 * part, because the values of 'rn' and 'next'
                                 * we compute will not be used. Not a big deal
                                 * because this loop will terminate, but it is
                                 * inefficient and hard to understand!
                                 */
                        }
                }

                /* Find the next *leaf* since next node might vanish, too */
                for (rn = rn->rn_parent->rn_right; rn->rn_bit >= 0;)
                        rn = rn->rn_left;
                next = rn;
                /* Process leaves */
                while ((rn = base) != 0) {
                        base = rn->rn_dupedkey;
                        /* printf("leaf %p\n", rn); */
                        if (!(rn->rn_flags & RNF_ROOT)
                            && (error = (*f)(rn, w)))
                                return (error);
                }
                rn = next;

                if (rn->rn_flags & RNF_ROOT) {
                        /* printf("root, stopping"); */
                        stopping = 1;
                }
```

```
        }
        return 0;
}

static int
rn_walktree(h, f, w)
        struct radix_node_head *h;
        walktree_f_t *f;
        void *w;
{
        int error;
        struct radix_node *base, *next;
        register struct radix_node *rn = h->rnh_treetop;
        /*
         * This gets complicated because we may delete the node
         * while applying the function f to it, so we need to calculate
         * the successor node in advance.
         */
        /* First time through node, go left */
        while (rn->rn_bit >= 0)
                rn = rn->rn_left;
        for (;;) {
                base = rn;
                /* If at right child go back up, otherwise, go right */
                while (rn->rn_parent->rn_right == rn
                       && (rn->rn_flags & RNF_ROOT) == 0)
                        rn = rn->rn_parent;
                /* Find the next *leaf* since next node might vanish, too */
                for (rn = rn->rn_parent->rn_right; rn->rn_bit >= 0;)
                        rn = rn->rn_left;
                next = rn;
                /* Process leaves */
                while ((rn = base)) {
                        base = rn->rn_dupedkey;
                        if (!(rn->rn_flags & RNF_ROOT)
                            && (error = (*f)(rn, w)))
                                return (error);
                }
                rn = next;
                if (rn->rn_flags & RNF_ROOT)
                        return (0);
        }
        /* NOTREACHED */
}

/*
 * Allocate and initialize an empty tree. This has 3 nodes, which are
 * part of the radix_node_head (in the order <left,root,right>) and are
 * marked RNF_ROOT so they cannot be freed.
 * The leaves have all-zero and all-one keys, with significant
 * bits starting at 'off'.
 * Return 1 on success, 0 on error.
 */
int
rn_inithead(head, off)
        void **head;
        int off;
{
        register struct radix_node_head *rnh;
        register struct radix_node *t, *tt, *ttt;
        if (*head)
                return (1);
        R_Zalloc(rnh, struct radix_node_head *, sizeof (*rnh));
        if (rnh == 0)
```

```
                return (0);
#ifdef _KERNEL
        RADIX_NODE_HEAD_LOCK_INIT(rnh);
#endif
        *head = rnh;
        t = rn_newpair(rn_zeros, off, rnh->rnh_nodes);
        ttt = rnh->rnh_nodes + 2;
        t->rn_right = ttt;
        t->rn_parent = t;
        tt = t->rn_left;        /* ... which in turn is rnh->rnh_nodes */
        tt->rn_flags = t->rn_flags = RNF_ROOT | RNF_ACTIVE;
        tt->rn_bit = -1 - off;
        *ttt = *tt;
        ttt->rn_key = rn_ones;
        rnh->rnh_addaddr = rn_addroute;
        rnh->rnh_deladdr = rn_delete;
        rnh->rnh_matchaddr = rn_match;
        rnh->rnh_lookup = rn_lookup;
        rnh->rnh_walktree = rn_walktree;
        rnh->rnh_walktree_from = rn_walktree_from;
        rnh->rnh_treetop = t;
        return (1);
}

void
rn_init()
{
        char *cp, *cplim;
#ifdef _KERNEL
        struct domain *dom;

        for (dom = domains; dom; dom = dom->dom_next)
                if (dom->dom_maxrtkey > max_keylen)
                        max_keylen = dom->dom_maxrtkey;
#endif
        if (max_keylen == 0) {
                log(LOG_ERR,
                    "rn_init: radix functions require max_keylen be set\n");
                return;
        }
        R_Malloc(rn_zeros, char *, 3 * max_keylen);
        if (rn_zeros == NULL)
                panic("rn_init");
        bzero(rn_zeros, 3 * max_keylen);
        rn_ones = cp = rn_zeros + max_keylen;
        addmask_key = cplim = rn_ones + max_keylen;
        while (cp < cplim)
                *cp++ = -1;
        if (rn_inithead((void **)(void *)&mask_rnhead, 0) == 0)
                panic("rn_init 2");
}
```

```
/*
 * Copyright (c) 1988, 1989, 1993
 *      The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 4. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *      @(#)radix.h     8.2 (Berkeley) 10/31/94
 * $FreeBSD: src/sys/net/radix.h,v 1.25 2004/04/18 11:48:35 luigi Exp $
 * $kbyanc: dyntrace/dyntrace/radix.h,v 1.2 2004/10/18 18:38:45 kbyanc Exp $
 */

#ifndef _RADIX_H_
#define _RADIX_H_

#ifdef _KERNEL
#include <sys/_lock.h>
#include <sys/_mutex.h>
#endif

#ifdef MALLOC_DECLARE
MALLOC_DECLARE(M_RTABLE);
#endif

/*
 * Radix search tree node layout.
 */

struct radix_node {
        struct  radix_mask *rn_mklist;  /* list of masks contained in subtree */
        struct  radix_node *rn_parent;  /* parent */
        short   rn_bit;                 /* bit offset; -1-index(netmask) */
        char    rn_bmask;               /* node: mask for bit test*/
        u_char  rn_flags;               /* enumerated next */
#define RNF_NORMAL      1               /* leaf contains normal route */
#define RNF_ROOT        2               /* leaf is root leaf for tree */
#define RNF_ACTIVE      4               /* This node is alive (for rtfree) */
        union {
                struct {                        /* leaf only data: */
                        caddr_t rn_Key;         /* object of search */
                        caddr_t rn_Mask;        /* netmask, if present */
                        struct  radix_node *rn_Dupedkey;
                } rn_leaf;
                struct {                        /* node only data: */
```

```
                        int     rn_Off;         /* where to start compare */
                        struct  radix_node *rn_L;/* progeny */
                        struct  radix_node *rn_R;/* progeny */
                } rn_node;
        }               rn_u;
#ifdef RN_DEBUG
        int rn_info;
        struct radix_node *rn_twin;
        struct radix_node *rn_ybro;
#endif
};

#define rn_dupedkey     rn_u.rn_leaf.rn_Dupedkey
#define rn_key          rn_u.rn_leaf.rn_Key
#define rn_mask         rn_u.rn_leaf.rn_Mask
#define rn_offset       rn_u.rn_node.rn_Off
#define rn_left         rn_u.rn_node.rn_L
#define rn_right        rn_u.rn_node.rn_R

/*
 * Annotations to tree concerning potential routes applying to subtrees.
 */

struct radix_mask {
        short   rm_bit;                 /* bit offset; -1-index(netmask) */
        char    rm_unused;              /* cf. rn_bmask */
        u_char  rm_flags;               /* cf. rn_flags */
        struct  radix_mask *rm_mklist;  /* more masks to try */
        union   {
                caddr_t rmu_mask;               /* the mask */
                struct  radix_node *rmu_leaf;   /* for normal routes */
        }       rm_rmu;
        int     rm_refs;                /* # of references to this struct */
};

#define rm_mask rm_rmu.rmu_mask
#define rm_leaf rm_rmu.rmu_leaf         /* extra field would make 32 bytes */

typedef int walktree_f_t(struct radix_node *, void *);

struct radix_node_head {
        struct  radix_node *rnh_treetop;
        int     rnh_addrsize;           /* permit, but not require fixed keys */
        int     rnh_pktsize;            /* permit, but not require fixed keys */
        struct  radix_node *(*rnh_addaddr)      /* add based on sockaddr */
                (void *v, void *mask,
                     struct radix_node_head *head, struct radix_node nodes[]);
        struct  radix_node *(*rnh_addpkt)       /* add based on packet hdr */
                (void *v, void *mask,
                     struct radix_node_head *head, struct radix_node nodes[]);
        struct  radix_node *(*rnh_deladdr)      /* remove based on sockaddr */
                (void *v, void *mask, struct radix_node_head *head);
        struct  radix_node *(*rnh_delpkt)       /* remove based on packet hdr */
                (void *v, void *mask, struct radix_node_head *head);
        struct  radix_node *(*rnh_matchaddr)    /* locate based on sockaddr */
                (void *v, struct radix_node_head *head);
        struct  radix_node *(*rnh_lookup)       /* locate based on sockaddr */
                (void *v, void *mask, struct radix_node_head *head);
        struct  radix_node *(*rnh_matchpkt)     /* locate based on packet hdr */
                (void *v, struct radix_node_head *head);
        int     (*rnh_walktree)                 /* traverse tree */
                (struct radix_node_head *head, walktree_f_t *f, void *w);
        int     (*rnh_walktree_from)            /* traverse tree below a */
                (struct radix_node_head *head, void *a, void *m,
                     walktree_f_t *f, void *w);
```

```
        void    (*rnh_close)    /* do something when the last ref drops */
                (struct radix_node *rn, struct radix_node_head *head);
        struct  radix_node rnh_nodes[3];        /* empty tree for common case */
#ifdef _KERNEL
        struct  mtx rnh_mtx;                    /* locks entire radix tree */
#endif
};

#ifndef _KERNEL
#define R_Malloc(p, t, n) (p = (t) malloc((unsigned int)(n)))
#define R_Zalloc(p, t, n) (p = (t) calloc(1,(unsigned int)(n)))
#define Free(p) free((char *)p);
#else
#define R_Malloc(p, t, n) (p = (t) malloc((unsigned long)(n), M_RTABLE, M_NOWAIT))
#define R_Zalloc(p, t, n) (p = (t) malloc((unsigned long)(n), M_RTABLE, M_NOWAIT | M_ZERO))
#define Free(p) free((caddr_t)p, M_RTABLE);

#define RADIX_NODE_HEAD_LOCK_INIT(rnh)  \
    mtx_init(&(rnh)->rnh_mtx, "radix node head", NULL, MTX_DEF | MTX_RECURSE)
#define RADIX_NODE_HEAD_LOCK(rnh)       mtx_lock(&(rnh)->rnh_mtx)
#define RADIX_NODE_HEAD_UNLOCK(rnh)     mtx_unlock(&(rnh)->rnh_mtx)
#define RADIX_NODE_HEAD_DESTROY(rnh)    mtx_destroy(&(rnh)->rnh_mtx)
#define RADIX_NODE_HEAD_LOCK_ASSERT(rnh) mtx_assert(&(rnh)->rnh_mtx, MA_OWNED)
#endif /* _KERNEL */

extern int max_keylen;

void     rn_init(void);
int      rn_inithead(void **, int);
int      rn_refines(void *, void *);
struct radix_node
         *rn_addmask(void *, int, int),
         *rn_addroute (void *, void *, struct radix_node_head *,
                       struct radix_node [2]),
         *rn_delete(void *, void *, struct radix_node_head *),
         *rn_lookup (void *v_arg, void *m_arg,
                       struct radix_node_head *head),
         *rn_match(void *, struct radix_node_head *);

#endif /* _RADIX_H_ */
```

```
/*
 * Copyright (c) 2004 Kelly Yancey
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $kbyanc: dyntrace/dyntrace/region.c,v 1.9 2004/12/27 04:31:54 kbyanc Exp $
 */

#include <sys/types.h>
#include <sys/queue.h>

#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <sysexits.h>

#include "dyntrace.h"


/*
 * Minimum and maximum number of bytes to cache per region of the target
 * process's address space.
 */
#define REGION_BUFFER_MINSIZE   32
#define REGION_BUFFER_MAXSIZE   1024*1024


struct region_info {
        LIST_ENTRY(region_info) link;

        vm_offset_t      start;
        vm_offset_t      end;

        region_type_t    type;
        bool             readonly;

        vm_offset_t      bufaddr;       /* First address cached. */
        size_t           buflen;        /* Bytes in cache buffer. */
        uint8_t         *buffer;
        size_t           bufsize;       /* Memory allocated to buffer. */
};


struct region_list {
        LIST_HEAD(, region_info) head;  /* List of regions. */
```

```
};


const char *region_type_name[NUMREGIONTYPES] = {
        "unknown",
        "text",
        "text:program",
        "text:library",
        "non-text",
        "data",
        "stack"
};

static region_t  region_find(region_list_t rlist, vm_offset_t addr);
static void      region_remove(region_t *regionp);


/*!
 * region_list_new() - Create a new region list.
 *
 *      @return a new region list.
 */
region_list_t
region_list_new(void)
{
        region_list_t rlist;

        rlist = malloc(sizeof(*rlist));
        if (rlist == NULL)
                fatal(EX_OSERR, "malloc: %m");

        LIST_INIT(&rlist->head);
        return rlist;
}


/*!
 * region_list_done() - Free all memory allocated to a region list.
 *
 *      @param  rlistp  Pointer to region list to free.
 *
 *      @post   The region list handle pointed to by *rlistp is invalidated.
 */
void
region_list_done(region_list_t *rlistp)
{
        region_list_t rlist = *rlistp;
        region_t region;

        *rlistp = NULL;

        while (!LIST_EMPTY(&rlist->head)) {
                region = LIST_FIRST(&rlist->head);
                region_remove(&region);
        }

        free(rlist);
}


/*!
 * region_find() - Internal routine to locate a region in a region list which
 *                 encloses the specified address.
 *
 *      @param  rlist   Region list to search.
```

```
 *
 *      @param  addr    The address to locate.
 *
 *      This is functionally identical to the region_lookup() routine except
 *      that it does not reorder to region list.  The intention is for
 *      region_find() to be used to locate regions without purturbing the list.
 */
region_t
region_find(region_list_t rlist, vm_offset_t addr)
{
        region_t region;

        LIST_FOREACH(region, &rlist->head, link) {
                if (region->start <= addr && region->end > addr)
                        return region;
        }

        return NULL;
}


/*!
 * region_lookup() - Locate the region in a region list which encloses the
 *                   specified address.
 *
 *      @param  rlist   Region list to search.
 *
 *      @param  addr    The address to locate.
 *
 *      Recently-accessed regions are moved to the head of the region list
 *      on the assumption they are most likely to be referenced again in
 *      the near future (due to locality of reference).
 */
region_t
region_lookup(region_list_t rlist, vm_offset_t addr)
{
        region_t region;

        region = region_find(rlist, addr);

        if (region == LIST_FIRST(&rlist->head) || region == NULL)
                return region;

        /*
         * Move the matched region to the head of the list to take advantage of
         * the locality of reference in the traced code.
         */
        LIST_REMOVE(region, link);
        LIST_INSERT_HEAD(&rlist->head, region, link);

        return region;
}


/*!
 * region_remove() - Remove a region from its region list and free it.
 *
 *      @param  regionp Pointer to region handle to remove.
 *
 *      @post   The region handle pointed to by regionp is invalidated.
 */
void
region_remove(region_t *regionp)
{
        region_t region = *regionp;
```

```
        *regionp = NULL;
        LIST_REMOVE(region, link);
        if (region->buffer != NULL)
                free(region->buffer);
        free(region);
}


/*!
 * region_update() - Update the given region list to include a region with
 *                   the specified properties.
 *
 *      @param  rlist   Region list to update.
 *
 *      @param  start   Memory region start address.
 *
 *      @param  end     Memory region end address.
 *
 *      @param  type    Type of memory region.
 *
 *      @param  readonly Whether or not the region is read-only.
 *
 *      Called from the system-specific memory map parser code to update
 *      the given region list.  Existing regions may be extended or replaced.
 */
void
region_update(region_list_t rlist, vm_offset_t start, vm_offset_t end,
              region_type_t type, bool readonly)
{
        region_t region;

        assert(end > start);

        /*
         * Lookup any existing regions which contain the new regions' start
         * address.  This will find overlapping regions, but not proper
         * sub-regions.  The latter is OK as the new region will be ahead
         * of the old region in the list so it will effectively "block" it.
         * This isn't ideal, but works as a time versus memory tradeoff.
         */
        while ((region = region_find(rlist, start)) != NULL) {

                /*
                 * If the new region exactly matches or is an extension of an
                 * existing region, then we simply update the existing region
                 * and return.  This is the most common case.
                 */
                if (region->start == start && region->end <= end &&
                    region->type == type && region->readonly == readonly) {
                        region->end = end;
                        return;
                }

                /*
                 * Remove any regions that overlap the start address.
                 */
                region_remove(&region);
        }

        assert(region == NULL);

        /*
         * Create a new region record and add it to the head of the list.
         */
```

```
        region = calloc(1, sizeof(*region));
        if (region == NULL)
                fatal(EX_OSERR, "malloc: %m");

        LIST_INSERT_HEAD(&rlist->head, region, link);

        region->start = start;
        region->end = end;
        region->type = type;
        region->readonly = readonly;

        if (!readonly)
                return;

        /*
         * The region is read-only so we can cache the memory contents to
         * save a call to the kernel for every instruction.  We cache the
         * minimum amount unless the region is a text segment, in which case
         * it is highly probable for code to be executed there so we cache
         * more.
         */
        region->bufsize = REGION_BUFFER_MINSIZE;

        if (REGION_IS_TEXT(region->type))
                region->bufsize = REGION_BUFFER_MAXSIZE;

        if (region->bufsize > end - start)
                region->bufsize = end - start;

        /*
         * Allocate buffer to cache the region's contents.  If the allocation
         * fails, just pretend the region isn't read-only.  This will likely
         * reduce throughput of the tracer, but will allow it to continue
         * to run without impacting the results.
         */
        region->buffer = malloc(region->bufsize);
        if (region->buffer == NULL) {
                warn("malloc: %m (non-fatal)");
                region->readonly = false;
        }
}


/*!
 * region_read() - Read contents of target process' memory utilizing the
 *                 region cache.
 *
 *      Reads the contents of the specified process' memory into a buffer
 *      in the current process.  If the region of memory being read is
 *      cacheable, the contents may be read from a cache and may be stored
 *      in the cache to satisfy future requests.
 *
 *      @param  targ    The target process whose memory contents to read.
 *
 *      @param  region  The target's memory region to read from.
 *
 *      @param  addr    Address within the target's virtual memory to read from.
 *                      This must be in the specified region.
 *
 *      @param  dest    Pointer to buffer to read contents into.
 *
 *      @param  len     The number of bytes to read.
 *
 *      @return number of bytes read.
```

```
 */
size_t
region_read(target_t targ, region_t region, vm_offset_t addr,
            void *dest, size_t len)
{
        vm_offset_t start;
        ssize_t offset;

        assert(len > 0);
        assert(addr + len <= region->end);

        /*
         * If the region is not readonly, we cannot cache the memory contents
         * as they may change (e.g. self-modifying code).  So we have to ask
         * the kernel to supply the memory contents every time.
         */
        if (!region->readonly)
                return target_read(targ, addr, dest, len);

        assert(region->buffer != NULL);

        offset = addr - region->bufaddr;

        /*
         * Satisfy the request from the region's cache if we can.
         */
        if (offset >= 0 && offset + len <= region->buflen) {
                memcpy(dest, region->buffer + offset, len);
                return len;
        }

        /*
         * Reload the region's cache.
         * We start the region cache slightly before the requested addr
         * so that simple loops do not cause spurious cache misses.
         */
        start = region->start;
        if (start + region->bufsize <= addr)
                start = region->end - region->bufsize;
        if (start > addr)
                start = addr + len - (region->bufsize / 2);

        region->buflen = region->end - start;
        if (region->buflen > region->bufsize)
                region->buflen = region->bufsize;

#if 0
        debug("XXX region cache miss, buflen = %u", region->buflen);
#endif

        region->buflen = target_read(targ, start, region->buffer,
                                     region->buflen);
        region->bufaddr = start;

        offset = addr - start;
        assert(offset >= 0);

        memcpy(dest, region->buffer + offset, len);
        return len;
}


/*!
 * region_get_type() - Get the type of a memory region.
 *
```

```
 *      @param  region  The memory region to get the type of.
 *
 *      @return region type code.
 */
region_type_t
region_get_type(region_t region)
{
        return region->type;
}


/*!
 * region_get_range() - Get the start and/or end addresses of a memory region.
 *
 *      @param  region  The memory region to get the start and/or end
 *                      addresses of.
 *
 *      @param  startp  Pointer to populate with the region's start address.
 *
 *      @param  endp    Pointer to populate with the region's end address.
 *
 *      @return the length of the region in bytes (i.e. difference between
 *              the region start and end addresses).
 *
 *      Either \a startp or \a endp can be NULL if the caller is not interested
 *      in the coresponding address.
 */
size_t
region_get_range(region_t region, vm_offset_t *startp, vm_offset_t *endp)
{

        assert(region != NULL);

        if (startp != NULL)
                *startp = region->start;
        if (endp != NULL)
                *endp = region->end;

        return (region->end - region->start);
}
```

```
/*
 * Copyright (c) 2004 Kelly Yancey
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $kbyanc: dyntrace/dyntrace/target_freebsd.c,v 1.8 2004/12/23 01:45:19 kbyanc Exp $
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/event.h>
#include <sys/sysctl.h>

#include <assert.h>
#include <inttypes.h>
#include <libgen.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <sysexits.h>
#include <time.h>
#include <unistd.h>

#if HAVE_PMC
#include <pmc.h>
#if __i386__
#include <machine/cpufunc.h>            /* for rdpmc() */
#endif
#endif

#include <machine/reg.h>

#include "dyntrace.h"
#include "procfs.h"
#include "ptrace.h"


struct target_state {
        pid_t            pid;           /* process identifier. */
        int              pfs_map;       /* procfs map file descriptor. */
        ptstate_t        pts;           /* ptrace(2) state. */
        region_list_t    rlist;         /* memory regions in process VM. */

#if HAVE_PMC
        pmc_id_t         pmc;           /* handle for PMC for cycle counts. */
```

```
        uint32_t         pmc_regnum;    /* x86 MSR number. */
        pmc_value_t      cycles;        /* cycle counter. */
#endif

        char            *procname;
};


static bool      pmc_avail = false;
static const char *pmc_eventname = NULL;

static vm_offset_t stack_top;
static int       kq;

/* Currently, we only support tracing a single process. */
static target_t  tracedproc = NULL;

static target_t  target_new(pid_t pid, ptstate_t pts, char *procname);
static void      target_region_refresh(target_t targ);
static void      freebsd_map_parseline(target_t targ, char *line, uint linenum);


void
target_init(void)
{
        size_t len;
        int mib[2];

        kq = kqueue();
        if (kq < 0)
                fatal(EX_OSERR, "kqueue: %m");

#if HAVE_PMC
        pmc_avail = (pmc_init() >= 0);
        if (pmc_avail) {
                const struct pmc_op_getcpuinfo *cpuinfo;

                if (pmc_cpuinfo(&cpuinfo) < 0)
                        fatal(EX_OSERR, "pmc_cpuinfo: %m");

                switch (cpuinfo->pm_cputype) {
                case PMC_CPU_INTEL_PIV:
                        pmc_eventname = "p4-global-power-events,usr";
                        break;

                case PMC_CPU_INTEL_PPRO:
                        pmc_eventname = "ppro-cpu-clk-unhalted,usr";
                        break;

                default:
                        pmc_avail = false;
                }
        }
#endif
        if (!pmc_avail)
                warn("pmc unavailable; instruction timing disabled");

        /*
         * FreeBSD always includes ptrace(2) support so we use for as much as
         * possible.  Currently, this includes process control, fetching
         * registers, and reading the target process's address space.
         */
        ptrace_init();

        /*
```

```
         * However, ptrace(2) does not provide a means to describe the target
         * process's address space.  For that, we use procfs.  However, there
         * is no guarantee that procfs is available; in which case we need to
         * warn the user that we cannot differentiate the type of the various
         * regions of the address space.
         * Note: The kvm(3) interface could be used instead as it is always
         *       available, but that runs the risk of breaking if the kernel
         *       data structures change.
         */
        if (!procfs_init())
                warn("procfs unavailable; region differentiation disabled");

        /*
         * Query the top-of-stack address.  We can use this information to
         * identify the main process stack in the process's region list.
         */
        mib[0] = CTL_KERN;
        mib[1] = KERN_USRSTACK;
        len = sizeof(stack_top);
        sysctl(mib, 2, &stack_top, &len, NULL, 0);
}


void
target_done(void)
{
}


target_t
target_new(pid_t pid, ptstate_t pts, char *procname)
{
        struct kevent kev;
        target_t targ;

        targ = calloc(1, sizeof(*targ));
        if (targ == NULL)
                fatal(EX_OSERR, "malloc: %m");

        targ->pid = pid;
        targ->pts = pts;
        targ->pfs_map = procfs_map_open(pid);
        targ->rlist = region_list_new();
        targ->procname = procname;

        assert(tracedproc == NULL);
        tracedproc = targ;

        target_region_refresh(targ);

        /*
         * Request notification whenever the process executes a new image.
         * This is necessary so we can flush the region cache.
         */
        EV_SET(&kev, pid, EVFILT_PROC, EV_ADD, NOTE_EXEC, 0, targ);
        if (kevent(kq, &kev, 1, NULL, 0, NULL) < 0)
                fatal(EX_OSERR, "kevent: %m");

#if HAVE_PMC
        if (pmc_avail) {
                if (pmc_allocate(strdup(pmc_eventname),
                                 PMC_MODE_TC, 0, PMC_CPU_ANY, &targ->pmc) < 0)
                        fatal(EX_OSERR, "pmc_allocate: %m");
                if (pmc_attach(targ->pmc, pid) < 0)
                        fatal(EX_OSERR, "pmc_attach: %m");
```

```
                if (pmc_rw(targ->pmc, 0, &targ->cycles) < 0)
                        fatal(EX_OSERR, "pmc_rw: %m");
#if __i386__
                /*
                 * On the x86 line of chips (Pentium and later) we can read
                 * the performance counter from userland using the rdpmc
                 * instruction, eliminating a context switch.  We just need
                 * to know which register our performance counter is in...
                 */
                if (pmc_i386_get_msr(targ->pmc, &targ->pmc_regnum) < 0)
                        fatal(EX_OSERR, "pmc_i386_get_msr: %m");
#endif
                if (pmc_start(targ->pmc) < 0)
                        fatal(EX_OSERR, "pmc_start: %m");
        }
#endif

        return targ;
}


target_t
target_execvp(const char *path, char * const argv[])
{
        char *procname;
        ptstate_t pts;
        pid_t pid;

        pts = ptrace_fork(&pid);
        if (pts == NULL) {
                /* Child process. */
                execvp(path, argv);
                fatal(EX_OSERR, "failed to execute \"%s\": %m", path);
        }

        procname = strdup(basename(path));
        if (procname == NULL)
                fatal(EX_OSERR, "malloc: %m");

        return target_new(pid, pts, procname);
}


target_t
target_attach(pid_t pid)
{
        char *procname;
        ptstate_t pts;

        pts = ptrace_attach(pid);

        /*
         * Try to use procfs to get the process name.  Failing that, fall back
         * to using the pid as the process name.
         */
        procname = procfs_get_procname(pid);
        if (procname == NULL)
                asprintf(&procname, "%u", pid);
        if (procname == NULL)
                fatal(EX_OSERR, "malloc: %m");

        return target_new(pid, pts, procname);
}
```

```
void
target_detach(target_t *targp)
{
        struct kevent kev;
        target_t targ = *targp;

        *targp = NULL;

#if HAVE_PMC
        if (pmc_avail) {
                pmc_stop(targ->pmc);
                pmc_release(targ->pmc);
        }
#endif

        EV_SET(&kev, targ->pid, EVFILT_PROC, EV_DELETE, NOTE_EXEC, 0, NULL);
        kevent(kq, &kev, 1, NULL, 0, NULL);     /* Not fatal if fails. */

        ptrace_detach(targ->pts);
        ptrace_done(&targ->pts);
        procfs_map_close(&targ->pfs_map);
        region_list_done(&targ->rlist);

        free(targ->procname);
        free(targ);

        tracedproc = NULL;
}


target_t
target_wait(void)
{
        static struct kevent events[1];
        static int nevents = 0;
        static struct kevent *kevp;
        static const struct timespec timeout = {0, 0};
        target_t targ;

        if (nevents == 0) {
                nevents = kevent(kq, NULL, 0, events, 1, &timeout);
                if (nevents < 0)
                        fatal(EX_OSERR, "kevent: %m");
                kevp = events;
        }

        if (nevents > 0) {
                assert(kevp->filter == EVFILT_PROC);

                targ = kevp->udata;
                assert(targ == tracedproc);

                if ((kevp->fflags & NOTE_EXEC) != 0) {
                        /*
                         * The traced process loaded a new process image so
                         * we need to invalidate the cache of the old image.
                         * Note that it is critical that we completely free
                         * the old region list and build a fresh one; just
                         * calling target_region_refresh() is not enough.
                         */
                        region_list_done(&targ->rlist);
                        targ->rlist = region_list_new();
                        target_region_refresh(targ);
                }
```

```
                kevp++;
                nevents--;
        }

        return ptrace_wait(tracedproc->pts) ? tracedproc : NULL;
}


void
target_step(target_t targ)
{
        ptrace_step(targ->pts);
}


size_t
target_read(target_t targ, vm_offset_t addr, void *dest, size_t len)
{
        return ptrace_read(targ->pts, addr, dest, len);
}


vm_offset_t
target_get_pc(target_t targ)
{
        struct reg regs;

        ptrace_getregs(targ->pts, &regs);
        return regs.r_eip;
}


uint
target_get_cycles(target_t targ)
{

#ifdef HAVE_PMC
        if (pmc_avail) {
                pmc_value_t cycles_prev = targ->cycles;

#if __i386__
                targ->cycles = rdpmc(targ->pmc_regnum);
#else
                if (pmc_read(targ->pmc, &targ->cycles) < 0)
                        fatal(EX_OSERR, "pmc_read: %m");
#endif
                assert(targ->cycles >= cycles_prev);
                return (targ->cycles - cycles_prev);
        }
#endif

        (void)targ;     /* Silence gcc warning when !HAVE_PMC. */
        return 0;
}


const char *
target_get_name(target_t targ)
{
        return targ->procname;
}


region_t
target_get_region(target_t targ, vm_offset_t addr)
```

```
{
        region_t region;

        region = region_lookup(targ->rlist, addr);
        if (region != NULL)
                return region;

        debug("refreshing region list; addr = 0x%08x", addr);

        target_region_refresh(targ);
        region = region_lookup(targ->rlist, addr);
        assert(region != NULL);
        return region;
}


void
target_region_refresh(target_t targ)
{
        char *pos, *endl;
        char *mapbuf;
        size_t maplen;
        uint linenum;

        if (targ->pfs_map < 0) {
                region_update(targ->rlist, 0, -1, REGION_UNKNOWN, false);
                return;
        }

        procfs_map_read(targ->pfs_map, &mapbuf, &maplen);
        assert(mapbuf != NULL);
        assert(mapbuf[maplen - 1] == '\n');

        linenum = 0;
        pos = mapbuf;
        while (maplen > 0) {
                endl = memchr(pos, '\n', maplen);
                if (endl == NULL)
                        break;
                *endl = '\0';

                freebsd_map_parseline(targ, pos, linenum);

                /* Advance to next line in map output. */
                linenum++;
                endl++;
                maplen -= endl - pos;
                pos = endl;
        }
}


void
freebsd_map_parseline(target_t targ, char *line, uint linenum)
{
        char *args[20];
        vm_offset_t start, end;
        region_type_t type;
        bool readonly;
        int i;

        memset(args, 0, sizeof(args));

        i = 0;
        while (i < 20 && (args[i] = strsep(&line, " \t")) != NULL) {
```

```
                if (*args[i] != '\0')
                        i++;
        }

        /* start = args[0]; */
        /* end   = args[1]; */
        /* perms = args[5]; (eg. rwx, r-x, ...) */
        /* type  = args[11]; (e.g. vnode) */
        /* path  = args[12]; */

        /* We aren't interested in regions that are not executable. */
        if (strchr(args[5], 'x') == NULL)
                return;

        readonly = (strchr(args[5], 'w') == NULL);

        start = strtoll(args[0], NULL, 16);
        end = strtoll(args[1], NULL, 16);

        type= REGION_NONTEXT_UNKNOWN;
        if (strcmp(args[11], "vnode") == 0) {
                if (linenum == 0)
                        type = REGION_TEXT_PROGRAM;
                else if (strcmp(args[5], "r-x") == 0)
                        type = REGION_TEXT_LIBRARY;
        }
        else if (end == stack_top)
                type = REGION_STACK;

        region_update(targ->rlist, start, end, type, readonly);
}
```