```
**********************************************************
   79290 Wed Oct 15 14:24:12 2014
new/usr/src/cmd/mdb/common/mdb/mdb_print.c
patch mdb-enums
**********************************************************
______unchanged_portion_omitted_

2539 static int
2540 printf_signed(mdb_ctf_id_t id, uintptr_t addr, ulong_t off, char *fmt,
2541     boolean_t sign)
2542 {
2543         ssize_t size;
2544         mdb_ctf_id_t base;
2545         ctf_encoding_t e;

2547         union {
2548                 uint64_t ui8;
2549                 uint32_t ui4;
2550                 uint16_t ui2;
2551                 uint8_t ui1;
2552                 int64_t i8;
2553                 int32_t i4;
2554                 int16_t i2;
2555                 int8_t i1;
2556         } u;

2558         if (mdb_ctf_type_resolve(id, &base) == -1) {
2559                 mdb_warn("could not resolve type");
2560                 return (DCMD_ABORT);
2561         }

2563         switch (mdb_ctf_type_kind(base)) {
2564                 case CTF_K_ENUM:
2565                         e.cte_format = CTF_INT_SIGNED;
2566                         e.cte_offset = 0;
2567                         e.cte_bits = sizeof(int) * NBBY; /* XXX: get size from C
2568                         break;
2569                 case CTF_K_INTEGER:
2570                         if (mdb_ctf_type_encoding(base, &e) != 0) {
2571                                 mdb_warn("could not get type encoding");
2563         if (mdb_ctf_type_kind(base) != CTF_K_INTEGER) {
2564                 mdb_warn("expected integer type\n");
2572                                 return (DCMD_ABORT);
2573                         }
2574                         break;
2575                 default:
2576                         mdb_warn("expected integer type\n");

2568         if (mdb_ctf_type_encoding(base, &e) != 0) {
2569                 mdb_warn("could not get type encoding");
2577                         return (DCMD_ABORT);
2578         }

2580         if (sign)
2581                 sign = e.cte_format & CTF_INT_SIGNED;

2583         size = e.cte_bits / NBBY;

2585         /*
2586          * Check to see if our life has been complicated by the presence of
2587          * a bitfield.  If it has, we will print it using logic that is only
2588          * slightly different than that found in print_bitfield(), above.  (In
2589          * particular, see the comments there for an explanation of the
2590          * endianness differences in this code.)
2591          */
2592         if (size > 8 || (e.cte_bits % NBBY) != 0 ||
```



```
2593             (size & (size - 1)) != 0) {
2594                 uint64_t mask = (1ULL << e.cte_bits) - 1;
2595                 uint64_t value = 0;
2596                 uint8_t *buf = (uint8_t *)&value;
2597                 uint8_t shift;

2599                 /*
2600                  * Round our size up one byte.
2601                  */
2602                 size = (e.cte_bits + (NBBY - 1)) / NBBY;

2604                 if (e.cte_bits > sizeof (value) * NBBY - 1) {
2605                         mdb_printf("invalid bitfield size %u", e.cte_bits);
2606                         return (DCMD_ABORT);
2607                 }

2609 #ifdef _BIG_ENDIAN
2610                 buf += sizeof (value) - size;
2611                 off += e.cte_bits;
2612 #endif

2614                 if (mdb_vread(buf, size, addr) == -1) {
2615                         mdb_warn("failed to read %lu bytes at %p", size, addr);
2616                         return (DCMD_ERR);
2617                 }

2619                 shift = off % NBBY;
2620 #ifdef _BIG_ENDIAN
2621                 shift = NBBY - shift;
2622 #endif

2624                 /*
2625                  * If we have a bit offset within the byte, shift it down.
2626                  */
2627                 if (off % NBBY != 0)
2628                         value >>= shift;
2629                 value &= mask;

2631                 if (sign) {
2632                         int sshift = sizeof (value) * NBBY - e.cte_bits;
2633                         value = ((int64_t)value << sshift) >> sshift;
2634                 }

2636                 mdb_printf(fmt, value);
2637                 return (0);
2638         }

2640         if (mdb_vread(&u.i8, size, addr) == -1) {
2641                 mdb_warn("failed to read %lu bytes at %p", (ulong_t)size, addr);
2642                 return (DCMD_ERR);
2643         }

2645         switch (size) {
2646         case sizeof (uint8_t):
2647                 mdb_printf(fmt, (uint64_t)(sign ? u.i1 : u.ui1));
2648                 break;
2649         case sizeof (uint16_t):
2650                 mdb_printf(fmt, (uint64_t)(sign ? u.i2 : u.ui2));
2651                 break;
2652         case sizeof (uint32_t):
2653                 mdb_printf(fmt, (uint64_t)(sign ? u.i4 : u.ui4));
2654                 break;
2655         case sizeof (uint64_t):
2656                 mdb_printf(fmt, (uint64_t)(sign ? u.i8 : u.ui8));
2657                 break;
2658         }
```

```
2660         return (0);
2661 }
_____unchanged_portion_omitted_

2732 /*ARGSUSED*/
2733 static int
2734 printf_string(mdb_ctf_id_t id, uintptr_t addr, ulong_t off, char *fmt)
2735 {
2736         mdb_ctf_id_t base;
2737         mdb_ctf_arinfo_t r;
2738         char buf[1024];
2739         ssize_t size;

2741         if (mdb_ctf_type_resolve(id, &base) == -1) {
2742                 mdb_warn("could not resolve type");
2743                 return (DCMD_ABORT);
2744         }

2746         if (mdb_ctf_type_kind(base) == CTF_K_POINTER) {
2747                 uintptr_t value;

2749                 if (mdb_vread(&value, sizeof (value), addr) == -1) {
2750                         mdb_warn("failed to read pointer at %llx", addr);
2751                         return (DCMD_ERR);
2752                 }

2754                 if (mdb_readstr(buf, sizeof (buf) - 1, value) < 0) {
2755                         mdb_warn("failed to read string at %llx", value);
2756                         return (DCMD_ERR);
2757                 }

2759                 mdb_printf(fmt, buf);
2760                 return (0);
2761         }

2763         if (mdb_ctf_type_kind(base) == CTF_K_ENUM) {
2764                 const char *strval;
2765                 int value;

2767                 if (mdb_vread(&value, sizeof (value), addr) == -1) {
2768                         mdb_warn("failed to read pointer at %llx", addr);
2769                         return (DCMD_ERR);
2770                 }

2772                 if ((strval = mdb_ctf_enum_name(id, value))) {
2773                         mdb_printf(fmt, strval);
2774                 } else {
2775                         mdb_snprintf(buf, sizeof (buf), "<%d>", value);
2776                         mdb_printf(fmt, buf);
2777                 }

2779                 return (DCMD_OK);
2780 #endif /* ! codereview */
2781         }

2783         if (mdb_ctf_type_kind(base) != CTF_K_ARRAY) {
2784                 mdb_warn("exepected pointer or array type\n");
2785                 return (DCMD_ABORT);
2786         }

2788         if (mdb_ctf_array_info(base, &r) == -1 ||
2789             mdb_ctf_type_resolve(r.mta_contents, &base) == -1 ||
2790             (size = mdb_ctf_type_size(base)) == -1) {
2791                 mdb_warn("can't determine array type");
2792                 return (DCMD_ABORT);
```



```
2793         }

2795         if (size != 1) {
2796                 mdb_warn("string format specifier requires "
2797                     "an array of characters\n");
2798                 return (DCMD_ABORT);
2799         }

2801         bzero(buf, sizeof (buf));

2803         if (mdb_vread(buf, MIN(r.mta_nelems, sizeof (buf) - 1), addr) == -1) {
2804                 mdb_warn("failed to read array at %p", addr);
2805                 return (DCMD_ERR);
2806         }

2808         mdb_printf(fmt, buf);

2810         return (0);
2811 }

2813 /*ARGSUSED*/
2814 static int
2815 printf_ipv6(mdb_ctf_id_t id, uintptr_t addr, ulong_t off, char *fmt)
2816 {
2817         mdb_ctf_id_t base;
2818         mdb_ctf_id_t ipv6_type, ipv6_base;
2819         in6_addr_t ipv6;

2821         if (mdb_ctf_lookup_by_name("in6_addr_t", &ipv6_type) == -1) {
2822                 mdb_warn("could not resolve in6_addr_t type\n");
2823                 return (DCMD_ABORT);
2824         }

2826         if (mdb_ctf_type_resolve(id, &base) == -1) {
2827                 mdb_warn("could not resolve type\n");
2828                 return (DCMD_ABORT);
2829         }

2831         if (mdb_ctf_type_resolve(ipv6_type, &ipv6_base) == -1) {
2832                 mdb_warn("could not resolve in6_addr_t type\n");
2833                 return (DCMD_ABORT);
2834         }

2836         if (mdb_ctf_type_cmp(base, ipv6_base) != 0) {
2837                 mdb_warn("requires argument of type in6_addr_t\n");
2838                 return (DCMD_ABORT);
2839         }

2841         if (mdb_vread(&ipv6, sizeof (ipv6), addr) == -1) {
2842                 mdb_warn("couldn't read in6_addr_t at %p", addr);
2843                 return (DCMD_ERR);
2844         }

2846         mdb_printf(fmt, &ipv6);

2848         return (0);
2849 }

2851 /*
2852  * To validate the format string specified to ::printf, we run the format
2853  * string through a very simple state machine that restricts us to a subset
2854  * of mdb_printf() functionality.
2855  */
2856 enum {
2857         PRINTF_NOFMT = 1,               /* no current format specifier */
2858         PRINTF_PERC,                    /* processed '%' */
```

```
2859         PRINTF_FMT,                    /* processing format specifier */
2860         PRINTF_LEFT,                   /* processed '-', expecting width */
2861         PRINTF_WIDTH,                  /* processing width */
2862         PRINTF_QUES                    /* processed '?', expecting format */
2863 };

2865 int
2866 cmd_printf_tab(mdb_tab_cookie_t *mcp, uint_t flags, int argc,
2867     const mdb_arg_t *argv)
2868 {
2869         int ii;
2870         char *f;

2872         /*
2873          * If argc doesn't have more than what should be the format string,
2874          * ignore it.
2875          */
2876         if (argc <= 1)
2877                 return (0);

2879         /*
2880          * Because we aren't leveraging the lex and yacc engine, we have to
2881          * manually walk the arguments to find both the first and last
2882          * open/close quote of the format string.
2883          */
2884         f = strchr(argv[0].a_un.a_str, '"');
2885         if (f == NULL)
2886                 return (0);

2888         f = strchr(f + 1, '"');
2889         if (f != NULL) {
2890                 ii = 0;
2891         } else {
2892                 for (ii = 1; ii < argc; ii++) {
2893                         if (argv[ii].a_type != MDB_TYPE_STRING)
2894                                 continue;
2895                         f = strchr(argv[ii].a_un.a_str, '"');
2896                         if (f != NULL)
2897                                 break;
2898                 }
2899                 /* Never found */
2900                 if (ii == argc)
2901                         return (0);
2902         }

2904         ii++;
2905         argc -= ii;
2906         argv += ii;

2908         return (cmd_print_tab_common(mcp, flags, argc, argv));
2909 }

2911 int
2912 cmd_printf(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
2913 {
2914         char type[MDB_SYM_NAMLEN];
2915         int i, nfmts = 0, ret;
2916         mdb_ctf_id_t id;
2917         const char *fmt, *member;
2918         char **fmts, *last, *dest, f;
2919         int (**funcs)(mdb_ctf_id_t, uintptr_t, ulong_t, char *);
2920         int state = PRINTF_NOFMT;
2921         printarg_t pa;

2923         if (!(flags & DCMD_ADDRSPEC))
2924                 return (DCMD_USAGE);
```



```
2926         bzero(&pa, sizeof (pa));
2927         pa.pa_as = MDB_TGT_AS_VIRT;
2928         pa.pa_realtgt = pa.pa_tgt = mdb.m_target;

2930         if (argc == 0 || argv[0].a_type != MDB_TYPE_STRING) {
2931                 mdb_warn("expected a format string\n");
2932                 return (DCMD_USAGE);
2933         }

2935         /*
2936          * Our first argument is a format string; rip it apart and run it
2937          * through our state machine to validate that our input is within the
2938          * subset of mdb_printf() format strings that we allow.
2939          */
2940         fmt = argv[0].a_un.a_str;
2941         /*
2942          * 'dest' must be large enough to hold a copy of the format string,
2943          * plus a NUL and up to 2 additional characters for each conversion
2944          * in the format string.  This gives us a bloat factor of 5/2 ~= 3.
2945          *   e.g. "%d" (strlen of 2) --> "%lld\0" (need 5 bytes)
2946          */
2947         dest = mdb_zalloc(strlen(fmt) * 3, UM_SLEEP | UM_GC);
2948         fmts = mdb_zalloc(strlen(fmt) * sizeof (char *), UM_SLEEP | UM_GC);
2949         funcs = mdb_zalloc(strlen(fmt) * sizeof (void *), UM_SLEEP | UM_GC);
2950         last = dest;

2952         for (i = 0; fmt[i] != '\0'; i++) {
2953                 *dest++ = f = fmt[i];

2955                 switch (state) {
2956                 case PRINTF_NOFMT:
2957                         state = f == '%' ? PRINTF_PERC : PRINTF_NOFMT;
2958                         break;

2960                 case PRINTF_PERC:
2961                         state = f == '-' ? PRINTF_LEFT :
2962                             f >= '0' && f <= '9' ? PRINTF_WIDTH :
2963                             f == '?' ? PRINTF_QUES :
2964                             f == '%' ? PRINTF_NOFMT : PRINTF_FMT;
2965                         break;

2967                 case PRINTF_LEFT:
2968                         state = f >= '0' && f <= '9' ? PRINTF_WIDTH :
2969                             f == '?' ? PRINTF_QUES : PRINTF_FMT;
2970                         break;

2972                 case PRINTF_WIDTH:
2973                         state = f >= '0' && f <= '9' ? PRINTF_WIDTH :
2974                             PRINTF_FMT;
2975                         break;

2977                 case PRINTF_QUES:
2978                         state = PRINTF_FMT;
2979                         break;
2980                 }

2982                 if (state != PRINTF_FMT)
2983                         continue;

2985                 dest--;

2987                 /*
2988                  * Now check that we have one of our valid format characters.
2989                  */
2990                 switch (f) {
```

```
2991                case 'a':
2992                case 'A':
2993                case 'p':
2994                        funcs[nfmts] = printf_ptr;
2995                        break;

2997                case 'd':
2998                case 'q':
2999                case 'R':
3000                        funcs[nfmts] = printf_int;
3001                        *dest++ = 'l';
3002                        *dest++ = 'l';
3003                        break;

3005                case 'I':
3006                        funcs[nfmts] = printf_uint32;
3007                        break;

3009                case 'N':
3010                        funcs[nfmts] = printf_ipv6;
3011                        break;

3013                case 'H':
3014                case 'o':
3015                case 'r':
3016                case 'u':
3017                case 'x':
3018                case 'X':
3019                        funcs[nfmts] = printf_uint;
3020                        *dest++ = 'l';
3021                        *dest++ = 'l';
3022                        break;

3024                case 's':
3025                        funcs[nfmts] = printf_string;
3026                        break;

3028                case 'Y':
3029                        funcs[nfmts] = sizeof (time_t) == sizeof (int) ?
3030                            printf_uint32 : printf_uint;
3031                        break;

3033                default:
3034                        mdb_warn("illegal format string at or near "
3035                            "'%c' (position %d)\n", f, i + 1);
3036                        return (DCMD_ABORT);
3037                }

3039                *dest++ = f;
3040                *dest++ = '\0';
3041                fmts[nfmts++] = last;
3042                last = dest;
3043                state = PRINTF_NOFMT;
3044        }

3046        argc--;
3047        argv++;

3049        /*
3050         * Now we expect a type name.
3051         */
3052        if ((ret = args_to_typename(&argc, &argv, type, sizeof (type))) != 0)
3053                return (ret);

3055        argv++;
3056        argc--;
```



```
3058        if (mdb_ctf_lookup_by_name(type, &id) != 0) {
3059                mdb_warn("failed to look up type %s", type);
3060                return (DCMD_ABORT);
3061        }

3063        if (argc == 0) {
3064                mdb_warn("at least one member must be specified\n");
3065                return (DCMD_USAGE);
3066        }

3068        if (argc != nfmts) {
3069                mdb_warn("%s format specifiers (found %d, expected %d)\n",
3070                    argc > nfmts ? "missing" : "extra", nfmts, argc);
3071                return (DCMD_ABORT);
3072        }

3074        for (i = 0; i < argc; i++) {
3075                mdb_ctf_id_t mid;
3076                ulong_t off;
3077                int ignored;

3079                if (argv[i].a_type != MDB_TYPE_STRING) {
3080                        mdb_warn("expected only type member arguments\n");
3081                        return (DCMD_ABORT);
3082                }

3084                if (strcmp((member = argv[i].a_un.a_str), ".") == 0) {
3085                        /*
3086                         * We allow "." to be specified to denote the current
3087                         * value of dot.
3088                         */
3089                        if (funcs[i] != printf_ptr && funcs[i] != printf_uint &&
3090                            funcs[i] != printf_int) {
3091                                mdb_warn("expected integer or pointer format "
3092                                    "specifier for '.'\n");
3093                                return (DCMD_ABORT);
3094                        }

3096                        mdb_printf(fmts[i], mdb_get_dot());
3097                        continue;
3098                }

3100                pa.pa_addr = addr;

3102                if (parse_member(&pa, member, id, &mid, &off, &ignored) != 0)
3103                        return (DCMD_ABORT);

3105                if ((ret = funcs[i](mid, pa.pa_addr, off, fmts[i])) != 0) {
3106                        mdb_warn("failed to print member '%s'\n", member);
3107                        return (ret);
3108                }
3109        }

3111        mdb_printf("%s", last);

3113        return (DCMD_OK);
3114 }

3116 static char _mdb_printf_help[] =
3117 "The format string argument is a printf(3C)-like format string that is a\n"
3118 "subset of the format strings supported by mdb_printf().  The type argument\n"
3119 "is the name of a type to be used to interpret the memory referenced by dot.\n"
3120 "The member should either be a field in the specified structure, or the\n"
3121 "special member '.', denoting the value of dot (and treated as a pointer).\n"
3122 "The number of members must match the number of format specifiers in the\n"
```

```
"format string.\n"
"\n"
"The following format specifiers are recognized by ::printf:\n"
"\n"
"  %%    Prints the '%' symbol.\n"
"  %a    Prints the member in symbolic form.\n"
"  %d    Prints the member as a decimal integer.  If the member is a signed\n"
"        integer type, the output will be signed.\n"
"  %H    Prints the member as a human-readable size.\n"
"  %I    Prints the member as an IPv4 address (must be 32-bit integer type).\n"
"  %N    Prints the member as an IPv6 address (must be of type in6_addr_t).\n"
"  %o    Prints the member as an unsigned octal integer.\n"
"  %p    Prints the member as a pointer, in hexadecimal.\n"
"  %q    Prints the member in signed octal.  Honk if you ever use this!\n"
"  %r    Prints the member as an unsigned value in the current output radix.\n"
"  %R    Prints the member as a signed value in the current output radix.\n"
"  %s    Prints the member as a string (requires a pointer or an array of\n"
"        characters).\n"
"  %u    Prints the member as an unsigned decimal integer.\n"
"  %x    Prints the member in hexadecimal.\n"
"  %X    Prints the member in hexadecimal, using the characters A-F as the\n"
"        digits for the values 10-15.\n"
"  %Y    Prints the member as a time_t as the string "
           "'year month day HH:MM:SS'.\n"
"\n"
"The following field width specifiers are recognized by ::printf:\n"
"\n"
"  %n    Field width is set to the specified decimal value.\n"
"  %?    Field width is set to the maximum width of a hexadecimal pointer\n"
"        value.  This is 8 in an ILP32 environment, and 16 in an LP64\n"
"        environment.\n"
"\n"
"The following flag specifers are recognized by ::printf:\n"
"\n"
"  %-    Left-justify the output within the specified field width.  If the\n"
"        width of the output is less than the specified field width, the\n"
"        output will be padded with blanks on the right-hand side.  Without\n"
"        %-, values are right-justified by default.\n"
"\n"
"  %0    Zero-fill the output field if the output is right-justified and the\n"
"        width of the output is less than the specified field width.  Without\n"
"        %0, right-justified values are prepended with blanks in order to\n"
"        fill the field.\n"
"\n"
"Examples: \n"
"\n"
"  ::walk proc | "
        "::printf \"%-6d %s\\n\" proc_t p_pidp->pid_id p_user.u_psargs\n"
"  ::walk thread | "
        "::printf \"%?p %3d %a\\n\" kthread_t . t_pri t_startpc\n"
"  ::walk zone | "
        "::printf \"%-40s %20s\\n\" zone_t zone_name zone_nodename\n"
"  ::walk ire | "
        "::printf \"%Y %I\\n\" ire_t ire_create_time ire_u.ire4_u.ire4_addr\n"
"\n";

void
printf_help(void)
{
        mdb_printf("%s", _mdb_printf_help);
}
```