```
/*
 * page-types: Tool for querying page flags
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the Free
 * Software Foundation; version 2.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should find a copy of v2 of the GNU General Public License somewhere on
 * your Linux system; if not, write to the Free Software Foundation, Inc., 59
 * Temple Place, Suite 330, Boston, MA 02111-1307 USA.
 *
 * Copyright (C) 2009 Intel corporation
 *
 * Authors: Wu Fengguang <fengguang.wu@intel.com>
 */

#define _LARGEFILE64_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <stdarg.h>
#include <string.h>
#include <getopt.h>
#include <limits.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/fcntl.h>


/*
 * pagemap kernel ABI bits
 */

#define PM_ENTRY_BYTES      sizeof(uint64_t)
#define PM_STATUS_BITS      3
#define PM_STATUS_OFFSET    (64 - PM_STATUS_BITS)
#define PM_STATUS_MASK      (((1LL << PM_STATUS_BITS) - 1) << PM_STATUS_OFFSET)
#define PM_STATUS(nr)       (((nr) << PM_STATUS_OFFSET) & PM_STATUS_MASK)
#define PM_PSHIFT_BITS      6
#define PM_PSHIFT_OFFSET    (PM_STATUS_OFFSET - PM_PSHIFT_BITS)
#define PM_PSHIFT_MASK      (((1LL << PM_PSHIFT_BITS) - 1) << PM_PSHIFT_OFFSET)
#define PM_PSHIFT(x)        (((u64) (x) << PM_PSHIFT_OFFSET) & PM_PSHIFT_MASK)
#define PM_PFRAME_MASK      ((1LL << PM_PSHIFT_OFFSET) - 1)
#define PM_PFRAME(x)        ((x) & PM_PFRAME_MASK)

#define PM_PRESENT          PM_STATUS(4LL)
#define PM_SWAP             PM_STATUS(2LL)


/*
 * kernel page flags
 */

#define KPF_BYTES       8
#define PROC_KPAGEFLAGS     "/proc/kpageflags"

/* copied from kpageflags_read() */
#define KPF_LOCKED      0
#define KPF_ERROR       1
```

```
#define KPF_REFERENCED      2
#define KPF_UPTODATE        3
#define KPF_DIRTY       4
#define KPF_LRU         5
#define KPF_ACTIVE      6
#define KPF_SLAB        7
#define KPF_WRITEBACK       8
#define KPF_RECLAIM     9
#define KPF_BUDDY       10

/* [11-20] new additions in 2.6.31 */
#define KPF_MMAP        11
#define KPF_ANON        12
#define KPF_SWAPCACHE       13
#define KPF_SWAPBACKED      14
#define KPF_COMPOUND_HEAD   15
#define KPF_COMPOUND_TAIL   16
#define KPF_HUGE        17
#define KPF_UNEVICTABLE     18
#define KPF_HWPOISON        19
#define KPF_NOPAGE      20
#define KPF_KSM         21

/* [32-] kernel hacking assistances */
#define KPF_RESERVED        32
#define KPF_MLOCKED     33
#define KPF_MAPPEDTODISK    34
#define KPF_PRIVATE     35
#define KPF_PRIVATE_2       36
#define KPF_OWNER_PRIVATE   37
#define KPF_ARCH        38
#define KPF_UNCACHED        39

/* [48-] take some arbitrary free slots for expanding overloaded flags
 * not part of kernel API
 */
#define KPF_READAHEAD       48
#define KPF_SLOB_FREE       49
#define KPF_SLUB_FROZEN     50
#define KPF_SLUB_DEBUG      51

#define KPF_ALL_BITS        ((uint64_t)~0ULL)
#define KPF_HACKERS_BITS    (0xffffULL << 32)
#define KPF_OVERLOADED_BITS (0xffffULL << 48)
#define BIT(name)       (1ULL << KPF_##name)
#define BITS_COMPOUND       (BIT(COMPOUND_HEAD) | BIT(COMPOUND_TAIL))

static const char *page_flag_names[] = {
    [KPF_LOCKED]        = "L:locked",
    [KPF_ERROR]     = "E:error",
    [KPF_REFERENCED]    = "R:referenced",
    [KPF_UPTODATE]      = "U:uptodate",
    [KPF_DIRTY]     = "D:dirty",
    [KPF_LRU]       = "l:lru",
    [KPF_ACTIVE]        = "A:active",
    [KPF_SLAB]      = "S:slab",
    [KPF_WRITEBACK]     = "W:writeback",
    [KPF_RECLAIM]       = "I:reclaim",
    [KPF_BUDDY]     = "B:buddy",

    [KPF_MMAP]      = "M:mmap",
    [KPF_ANON]      = "a:anonymous",
    [KPF_SWAPCACHE]     = "s:swapcache",
    [KPF_SWAPBACKED]    = "b:swapbacked",
    [KPF_COMPOUND_HEAD] = "H:compound_head",
    [KPF_COMPOUND_TAIL] = "T:compound_tail",
```

```
    [KPF_HUGE]      = "G:huge",
    [KPF_UNEVICTABLE]   = "u:unevictable",
    [KPF_HWPOISON]      = "X:hwpoison",
    [KPF_NOPAGE]        = "n:nopage",
    [KPF_KSM]       = "x:ksm",

    [KPF_RESERVED]      = "r:reserved",
    [KPF_MLOCKED]       = "m:mlocked",
    [KPF_MAPPEDTODISK]  = "d:mappedtodisk",
    [KPF_PRIVATE]       = "P:private",
    [KPF_PRIVATE_2]     = "p:private_2",
    [KPF_OWNER_PRIVATE] = "O:owner_private",
    [KPF_ARCH]      = "h:arch",
    [KPF_UNCACHED]      = "c:uncached",

    [KPF_READAHEAD]     = "I:readahead",
    [KPF_SLOB_FREE]     = "P:slob_free",
    [KPF_SLUB_FROZEN]   = "A:slub_frozen",
    [KPF_SLUB_DEBUG]    = "E:slub_debug",
};


/*
 * data structures
 */

static int      opt_raw;    /* for kernel developers */
static int      opt_list;   /* list pages (in ranges) */
static int      opt_no_summary; /* don't show summary */
static pid_t        opt_pid;    /* process to walk */

#define MAX_ADDR_RANGES 1024
static int      nr_addr_ranges;
static unsigned long    opt_offset[MAX_ADDR_RANGES];
static unsigned long    opt_size[MAX_ADDR_RANGES];

#define MAX_VMAS    10240
static int      nr_vmas;
static unsigned long    pg_start[MAX_VMAS];
static unsigned long    pg_end[MAX_VMAS];

#define MAX_BIT_FILTERS 64
static int      nr_bit_filters;
static uint64_t     opt_mask[MAX_BIT_FILTERS];
static uint64_t     opt_bits[MAX_BIT_FILTERS];

static int      page_size;

static int      pagemap_fd;
static int      kpageflags_fd;

static int      opt_hwpoison;
static int      opt_unpoison;

static const char   hwpoison_debug_fs[] = "/debug/hwpoison";
static int      hwpoison_inject_fd;
static int      hwpoison_forget_fd;

#define HASH_SHIFT  13
#define HASH_SIZE   (1 << HASH_SHIFT)
#define HASH_MASK   (HASH_SIZE - 1)
#define HASH_KEY(flags) (flags & HASH_MASK)

static unsigned long    total_pages;
static unsigned long    nr_pages[HASH_SIZE];
static uint64_t     page_flags[HASH_SIZE];
```

```
/*
 * helper functions
 */

#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))

#define min_t(type, x, y) ({            \
    type __min1 = (x);          \
    type __min2 = (y);          \
    __min1 < __min2 ? __min1 : __min2; })

#define max_t(type, x, y) ({            \
    type __max1 = (x);          \
    type __max2 = (y);          \
    __max1 > __max2 ? __max1 : __max2; })

static unsigned long pages2mb(unsigned long pages)
{
    return (pages * page_size) >> 20;
}

static void fatal(const char *x, ...)
{
    va_list ap;

    va_start(ap, x);
    vfprintf(stderr, x, ap);
    va_end(ap);
    exit(EXIT_FAILURE);
}

static int checked_open(const char *pathname, int flags)
{
    int fd = open(pathname, flags);

    if (fd < 0) {
        perror(pathname);
        exit(EXIT_FAILURE);
    }

    return fd;
}

/*
 * pagemap/kpageflags routines
 */

static unsigned long do_u64_read(int fd, char *name,
                 uint64_t *buf,
                 unsigned long index,
                 unsigned long count)
{
    long bytes;

    if (index > ULONG_MAX / 8)
        fatal("index overflow: %lu\n", index);

    if (lseek(fd, index * 8, SEEK_SET) < 0) {
        perror(name);
        exit(EXIT_FAILURE);
    }

    bytes = read(fd, buf, count * 8);
    if (bytes < 0) {
```

```
        perror(name);
        exit(EXIT_FAILURE);
    }
    if (bytes % 8)
        fatal("partial read: %lu bytes\n", bytes);

    return bytes / 8;
}

static unsigned long kpageflags_read(uint64_t *buf,
                     unsigned long index,
                     unsigned long pages)
{
    return do_u64_read(kpageflags_fd, PROC_KPAGEFLAGS, buf, index, pages);
}

static unsigned long pagemap_read(uint64_t *buf,
                  unsigned long index,
                  unsigned long pages)
{
    return do_u64_read(pagemap_fd, "/proc/pid/pagemap", buf, index, pages);
}

static unsigned long pagemap_pfn(uint64_t val)
{
    unsigned long pfn;

    if (val & PM_PRESENT)
        pfn = PM_PFRAME(val);
    else
        pfn = 0;

    return pfn;
}


/*
 * page flag names
 */

static char *page_flag_name(uint64_t flags)
{
    static char buf[65];
    int present;
    int i, j;

    for (i = 0, j = 0; i < ARRAY_SIZE(page_flag_names); i++) {
        present = (flags >> i) & 1;
        if (!page_flag_names[i]) {
            if (present)
                fatal("unknown flag bit %d\n", i);
            continue;
        }
        buf[j++] = present ? page_flag_names[i][0] : '_';
    }

    return buf;
}

static char *page_flag_longname(uint64_t flags)
{
    static char buf[1024];
    int i, n;

    for (i = 0, n = 0; i < ARRAY_SIZE(page_flag_names); i++) {
        if (!page_flag_names[i])
```

```
            continue;
        if ((flags >> i) & 1)
            n += snprintf(buf + n, sizeof(buf) - n, "%s,",
                    page_flag_names[i] + 2);
    }
    if (n)
        n--;
    buf[n] = '\0';

    return buf;
}


/*
 * page list and summary
 */

static void show_page_range(unsigned long voffset,
                unsigned long offset, uint64_t flags)
{
    static uint64_t      flags0;
    static unsigned long voff;
    static unsigned long index;
    static unsigned long count;

    if (flags == flags0 && offset == index + count &&
        (!opt_pid || voffset == voff + count)) {
        count++;
        return;
    }

    if (count) {
        if (opt_pid)
            printf("%lx\t", voff);
        printf("%lx\t%lx\t%s\n",
                index, count, page_flag_name(flags0));
    }

    flags0 = flags;
    index  = offset;
    voff   = voffset;
    count  = 1;
}

static void show_page(unsigned long voffset,
              unsigned long offset, uint64_t flags)
{
    if (opt_pid)
        printf("%lx\t", voffset);
    printf("%lx\t%s\n", offset, page_flag_name(flags));
}

static void show_summary(void)
{
    int i;

    printf("             flags\tpage-count       MB"
        "  symbolic-flags\t\t\tlong-symbolic-flags\n");

    for (i = 0; i < ARRAY_SIZE(nr_pages); i++) {
        if (nr_pages[i])
            printf("0x%016llx\t%10lu %8lu  %s\t%s\n",
                (unsigned long long)page_flags[i],
                nr_pages[i],
                pages2mb(nr_pages[i]),
                page_flag_name(page_flags[i]),
```

```
                page_flag_longname(page_flags[i]));
    }

    printf("             total\t%10lu %8lu\n",
            total_pages, pages2mb(total_pages));
}


/*
 * page flag filters
 */

static int bit_mask_ok(uint64_t flags)
{
    int i;

    for (i = 0; i < nr_bit_filters; i++) {
        if (opt_bits[i] == KPF_ALL_BITS) {
            if ((flags & opt_mask[i]) == 0)
                return 0;
        } else {
            if ((flags & opt_mask[i]) != opt_bits[i])
                return 0;
        }
    }

    return 1;
}

static uint64_t expand_overloaded_flags(uint64_t flags)
{
    /* SLOB/SLUB overload several page flags */
    if (flags & BIT(SLAB)) {
        if (flags & BIT(PRIVATE))
            flags ^= BIT(PRIVATE) | BIT(SLOB_FREE);
        if (flags & BIT(ACTIVE))
            flags ^= BIT(ACTIVE) | BIT(SLUB_FROZEN);
        if (flags & BIT(ERROR))
            flags ^= BIT(ERROR) | BIT(SLUB_DEBUG);
    }

    /* PG_reclaim is overloaded as PG_readahead in the read path */
    if ((flags & (BIT(RECLAIM) | BIT(WRITEBACK))) == BIT(RECLAIM))
        flags ^= BIT(RECLAIM) | BIT(READAHEAD);

    return flags;
}

static uint64_t well_known_flags(uint64_t flags)
{
    /* hide flags intended only for kernel hacker */
    flags &= ~KPF_HACKERS_BITS;

    /* hide non-hugeTLB compound pages */
    if ((flags & BITS_COMPOUND) && !(flags & BIT(HUGE)))
        flags &= ~BITS_COMPOUND;

    return flags;
}

static uint64_t kpageflags_flags(uint64_t flags)
{
    flags = expand_overloaded_flags(flags);

    if (!opt_raw)
        flags = well_known_flags(flags);
```

```
    return flags;
}

/*
 * page actions
 */

static void prepare_hwpoison_fd(void)
{
    char buf[100];

    if (opt_hwpoison && !hwpoison_inject_fd) {
        sprintf(buf, "%s/corrupt-pfn", hwpoison_debug_fs);
        hwpoison_inject_fd = checked_open(buf, O_WRONLY);
    }

    if (opt_unpoison && !hwpoison_forget_fd) {
        sprintf(buf, "%s/renew-pfn", hwpoison_debug_fs);
        hwpoison_forget_fd = checked_open(buf, O_WRONLY);
    }
}

static int hwpoison_page(unsigned long offset)
{
    char buf[100];
    int len;

    len = sprintf(buf, "0x%lx\n", offset);
    len = write(hwpoison_inject_fd, buf, len);
    if (len < 0) {
        perror("hwpoison inject");
        return len;
    }
    return 0;
}

static int unpoison_page(unsigned long offset)
{
    char buf[100];
    int len;

    len = sprintf(buf, "0x%lx\n", offset);
    len = write(hwpoison_forget_fd, buf, len);
    if (len < 0) {
        perror("hwpoison forget");
        return len;
    }
    return 0;
}

/*
 * page frame walker
 */

static int hash_slot(uint64_t flags)
{
    int k = HASH_KEY(flags);
    int i;

    /* Explicitly reserve slot 0 for flags 0: the following logic
     * cannot distinguish an unoccupied slot from slot (flags==0).
     */
    if (flags == 0)
        return 0;
```

```
    /* search through the remaining (HASH_SIZE-1) slots */
    for (i = 1; i < ARRAY_SIZE(page_flags); i++, k++) {
        if (!k || k >= ARRAY_SIZE(page_flags))
            k = 1;
        if (page_flags[k] == 0) {
            page_flags[k] = flags;
            return k;
        }
        if (page_flags[k] == flags)
            return k;
    }

    fatal("hash table full: bump up HASH_SHIFT?\n");
    exit(EXIT_FAILURE);
}

static void add_page(unsigned long voffset,
             unsigned long offset, uint64_t flags)
{
    flags = kpageflags_flags(flags);

    if (!bit_mask_ok(flags))
        return;

    if (opt_hwpoison)
        hwpoison_page(offset);
    if (opt_unpoison)
        unpoison_page(offset);

    if (opt_list == 1)
        show_page_range(voffset, offset, flags);
    else if (opt_list == 2)
        show_page(voffset, offset, flags);

    nr_pages[hash_slot(flags)]++;
    total_pages++;
}

#define KPAGEFLAGS_BATCH    (64 << 10)  /* 64k pages */
static void walk_pfn(unsigned long voffset,
             unsigned long index,
             unsigned long count)
{
    uint64_t buf[KPAGEFLAGS_BATCH];
    unsigned long batch;
    long pages;
    unsigned long i;

    while (count) {
        batch = min_t(unsigned long, count, KPAGEFLAGS_BATCH);
        pages = kpageflags_read(buf, index, batch);
        if (pages == 0)
            break;

        for (i = 0; i < pages; i++)
            add_page(voffset + i, index + i, buf[i]);

        index += pages;
        count -= pages;
    }
}

#define PAGEMAP_BATCH   (64 << 10)
static void walk_vma(unsigned long index, unsigned long count)
{
    uint64_t buf[PAGEMAP_BATCH];
```

```
    unsigned long batch;
    unsigned long pages;
    unsigned long pfn;
    unsigned long i;

    while (count) {
        batch = min_t(unsigned long, count, PAGEMAP_BATCH);
        pages = pagemap_read(buf, index, batch);
        if (pages == 0)
            break;

        for (i = 0; i < pages; i++) {
            pfn = pagemap_pfn(buf[i]);
            if (pfn)
                walk_pfn(index + i, pfn, 1);
        }

        index += pages;
        count -= pages;
    }
}

static void walk_task(unsigned long index, unsigned long count)
{
    const unsigned long end = index + count;
    unsigned long start;
    int i = 0;

    while (index < end) {

        while (pg_end[i] <= index)
            if (++i >= nr_vmas)
                return;
        if (pg_start[i] >= end)
            return;

        start = max_t(unsigned long, pg_start[i], index);
        index = min_t(unsigned long, pg_end[i], end);

        assert(start < index);
        walk_vma(start, index - start);
    }
}

static void add_addr_range(unsigned long offset, unsigned long size)
{
    if (nr_addr_ranges >= MAX_ADDR_RANGES)
        fatal("too many addr ranges\n");

    opt_offset[nr_addr_ranges] = offset;
    opt_size[nr_addr_ranges] = min_t(unsigned long, size, ULONG_MAX-offset);
    nr_addr_ranges++;
}

static void walk_addr_ranges(void)
{
    int i;

    kpageflags_fd = checked_open(PROC_KPAGEFLAGS, O_RDONLY);

    if (!nr_addr_ranges)
        add_addr_range(0, ULONG_MAX);

    for (i = 0; i < nr_addr_ranges; i++)
        if (!opt_pid)
            walk_pfn(0, opt_offset[i], opt_size[i]);
```

```
        else
            walk_task(opt_offset[i], opt_size[i]);

    close(kpageflags_fd);
}


/*
 * user interface
 */

static const char *page_flag_type(uint64_t flag)
{
    if (flag & KPF_HACKERS_BITS)
        return "(r)";
    if (flag & KPF_OVERLOADED_BITS)
        return "(o)";
    return "   ";
}

static void usage(void)
{
    int i, j;

    printf(
"page-types [options]\n"
"            -r|--raw                  Raw mode, for kernel developers\n"
"            -d|--describe flags       Describe flags\n"
"            -a|--addr    addr-spec    Walk a range of pages\n"
"            -b|--bits    bits-spec    Walk pages with specified bits\n"
"            -p|--pid     pid          Walk process address space\n"
#if 0 /* planned features */
"            -f|--file    filename     Walk file address space\n"
#endif
"            -l|--list                 Show page details in ranges\n"
"            -L|--list-each            Show page details one by one\n"
"            -N|--no-summary           Don't show summay info\n"
"            -X|--hwpoison             hwpoison pages\n"
"            -x|--unpoison             unpoison pages\n"
"            -h|--help                 Show this usage message\n"
"flags:\n"
"            0x10                      bitfield format, e.g.\n"
"            anon                      bit-name, e.g.\n"
"            0x10,anon                 comma-separated list, e.g.\n"
"addr-spec:\n"
"            N                         one page at offset N (unit: pages)\n"
"            N+M                       pages range from N to N+M-1\n"
"            N,M                       pages range from N to M-1\n"
"            N,                        pages range from N to end\n"
"            ,M                        pages range from 0 to M-1\n"
"bits-spec:\n"
"            bit1,bit2                 (flags & (bit1|bit2)) != 0\n"
"            bit1,bit2=bit1            (flags & (bit1|bit2)) == bit1\n"
"            bit1,~bit2                (flags & (bit1|bit2)) == bit1\n"
"            =bit1,bit2                flags == (bit1|bit2)\n"
"bit-names:\n"
    );

    for (i = 0, j = 0; i < ARRAY_SIZE(page_flag_names); i++) {
        if (!page_flag_names[i])
            continue;
        printf("%16s%s", page_flag_names[i] + 2,
                page_flag_type(1ULL << i));
        if (++j > 3) {
            j = 0;
            putchar('\n');
```

```
        }
    }
    printf("\n                                    "
        "(r) raw mode bits  (o) overloaded bits\n");
}

static unsigned long long parse_number(const char *str)
{
    unsigned long long n;

    n = strtoll(str, NULL, 0);

    if (n == 0 && str[0] != '0')
        fatal("invalid name or number: %s\n", str);

    return n;
}

static void parse_pid(const char *str)
{
    FILE *file;
    char buf[5000];

    opt_pid = parse_number(str);

    sprintf(buf, "/proc/%d/pagemap", opt_pid);
    pagemap_fd = checked_open(buf, O_RDONLY);

    sprintf(buf, "/proc/%d/maps", opt_pid);
    file = fopen(buf, "r");
    if (!file) {
        perror(buf);
        exit(EXIT_FAILURE);
    }

    while (fgets(buf, sizeof(buf), file) != NULL) {
        unsigned long vm_start;
        unsigned long vm_end;
        unsigned long long pgoff;
        int major, minor;
        char r, w, x, s;
        unsigned long ino;
        int n;

        n = sscanf(buf, "%lx-%lx %c%c%c%c %llx %x:%x %lu",
               &vm_start,
               &vm_end,
               &r, &w, &x, &s,
               &pgoff,
               &major, &minor,
               &ino);
        if (n < 10) {
            fprintf(stderr, "unexpected line: %s\n", buf);
            continue;
        }
        pg_start[nr_vmas] = vm_start / page_size;
        pg_end[nr_vmas] = vm_end / page_size;
        if (++nr_vmas >= MAX_VMAS) {
            fprintf(stderr, "too many VMAs\n");
            break;
        }
    }
    fclose(file);
}

static void parse_file(const char *name)
```

```
{
}

static void parse_addr_range(const char *optarg)
{
    unsigned long offset;
    unsigned long size;
    char *p;

    p = strchr(optarg, ',');
    if (!p)
        p = strchr(optarg, '+');

    if (p == optarg) {
        offset = 0;
        size   = parse_number(p + 1);
    } else if (p) {
        offset = parse_number(optarg);
        if (p[1] == '\0')
            size = ULONG_MAX;
        else {
            size = parse_number(p + 1);
            if (*p == ',') {
                if (size < offset)
                    fatal("invalid range: %lu,%lu\n",
                            offset, size);
                size -= offset;
            }
        }
    } else {
        offset = parse_number(optarg);
        size   = 1;
    }

    add_addr_range(offset, size);
}

static void add_bits_filter(uint64_t mask, uint64_t bits)
{
    if (nr_bit_filters >= MAX_BIT_FILTERS)
        fatal("too much bit filters\n");

    opt_mask[nr_bit_filters] = mask;
    opt_bits[nr_bit_filters] = bits;
    nr_bit_filters++;
}

static uint64_t parse_flag_name(const char *str, int len)
{
    int i;

    if (!*str || !len)
        return 0;

    if (len <= 8 && !strncmp(str, "compound", len))
        return BITS_COMPOUND;

    for (i = 0; i < ARRAY_SIZE(page_flag_names); i++) {
        if (!page_flag_names[i])
            continue;
        if (!strncmp(str, page_flag_names[i] + 2, len))
            return 1ULL << i;
    }

    return parse_number(str);
}
```

```
static uint64_t parse_flag_names(const char *str, int all)
{
    const char *p    = str;
    uint64_t   flags = 0;

    while (1) {
        if (*p == ',' || *p == '=' || *p == '\0') {
            if ((*str != '~') || (*str == '~' && all && *++str))
                flags |= parse_flag_name(str, p - str);
            if (*p != ',')
                break;
            str = p + 1;
        }
        p++;
    }

    return flags;
}

static void parse_bits_mask(const char *optarg)
{
    uint64_t mask;
    uint64_t bits;
    const char *p;

    p = strchr(optarg, '=');
    if (p == optarg) {
        mask = KPF_ALL_BITS;
        bits = parse_flag_names(p + 1, 0);
    } else if (p) {
        mask = parse_flag_names(optarg, 0);
        bits = parse_flag_names(p + 1, 0);
    } else if (strchr(optarg, '~')) {
        mask = parse_flag_names(optarg, 1);
        bits = parse_flag_names(optarg, 0);
    } else {
        mask = parse_flag_names(optarg, 0);
        bits = KPF_ALL_BITS;
    }

    add_bits_filter(mask, bits);
}

static void describe_flags(const char *optarg)
{
    uint64_t flags = parse_flag_names(optarg, 0);

    printf("0x%016llx\t%s\t%s\n",
        (unsigned long long)flags,
        page_flag_name(flags),
        page_flag_longname(flags));
}

static const struct option opts[] = {
    { "raw"       , 0, NULL, 'r' },
    { "pid"       , 1, NULL, 'p' },
    { "file"      , 1, NULL, 'f' },
    { "addr"      , 1, NULL, 'a' },
    { "bits"      , 1, NULL, 'b' },
    { "describe"  , 1, NULL, 'd' },
    { "list"      , 0, NULL, 'l' },
    { "list-each" , 0, NULL, 'L' },
    { "no-summary", 0, NULL, 'N' },
    { "hwpoison"  , 0, NULL, 'X' },
    { "unpoison"  , 0, NULL, 'x' },
```

```
    { "help"      , 0, NULL, 'h' },
    { NULL        , 0, NULL, 0 }
};

int main(int argc, char *argv[])
{
    int c;

    page_size = getpagesize();

    while ((c = getopt_long(argc, argv,
                "rp:f:a:b:d:lLNXxh", opts, NULL)) != -1) {
        switch (c) {
        case 'r':
            opt_raw = 1;
            break;
        case 'p':
            parse_pid(optarg);
            break;
        case 'f':
            parse_file(optarg);
            break;
        case 'a':
            parse_addr_range(optarg);
            break;
        case 'b':
            parse_bits_mask(optarg);
            break;
        case 'd':
            describe_flags(optarg);
            exit(0);
        case 'l':
            opt_list = 1;
            break;
        case 'L':
            opt_list = 2;
            break;
        case 'N':
            opt_no_summary = 1;
            break;
        case 'X':
            opt_hwpoison = 1;
            prepare_hwpoison_fd();
            break;
        case 'x':
            opt_unpoison = 1;
            prepare_hwpoison_fd();
            break;
        case 'h':
            usage();
            exit(0);
        default:
            usage();
            exit(1);
        }
    }

    if (opt_list && opt_pid)
        printf("voffset\t");
    if (opt_list == 1)
        printf("offset\tlen\tflags\n");
    if (opt_list == 2)
        printf("offset\tflags\n");

    walk_addr_ranges();
```

```
    if (opt_list == 1)
        show_page_range(0, 0, 0);  /* drain the buffer */

    if (opt_no_summary)
        return 0;

    if (opt_list)
        printf("\n\n");

    show_summary();

    return 0;
}
```