```
/*
 * Slabinfo: Tool to get reports about slabs
 *
 * (C) 2007 sgi, Christoph Lameter
 *
 * Compile by:
 *
 * gcc -o slabinfo slabinfo.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>
#include <strings.h>
#include <string.h>
#include <unistd.h>
#include <stdarg.h>
#include <getopt.h>
#include <regex.h>
#include <errno.h>

#define MAX_SLABS 500
#define MAX_ALIASES 500
#define MAX_NODES 1024

struct slabinfo {
    char *name;
    int alias;
    int refs;
    int aliases, align, cache_dma, cpu_slabs, destroy_by_rcu;
    int hwcache_align, object_size, objs_per_slab;
    int sanity_checks, slab_size, store_user, trace;
    int order, poison, reclaim_account, red_zone;
    unsigned long partial, objects, slabs, objects_partial, objects_total;
    unsigned long alloc_fastpath, alloc_slowpath;
    unsigned long free_fastpath, free_slowpath;
    unsigned long free_frozen, free_add_partial, free_remove_partial;
    unsigned long alloc_from_partial, alloc_slab, free_slab, alloc_refill;
    unsigned long cpuslab_flush, deactivate_full, deactivate_empty;
    unsigned long deactivate_to_head, deactivate_to_tail;
    unsigned long deactivate_remote_frees, order_fallback;
    int numa[MAX_NODES];
    int numa_partial[MAX_NODES];
} slabinfo[MAX_SLABS];

struct aliasinfo {
    char *name;
    char *ref;
    struct slabinfo *slab;
} aliasinfo[MAX_ALIASES];

int slabs = 0;
int actual_slabs = 0;
int aliases = 0;
int alias_targets = 0;
int highest_node = 0;

char buffer[4096];

int show_empty = 0;
int show_report = 0;
int show_alias = 0;
int show_slab = 0;
int skip_zero = 1;
int show_numa = 0;
int show_track = 0;
```

```
int show_first_alias = 0;
int validate = 0;
int shrink = 0;
int show_inverted = 0;
int show_single_ref = 0;
int show_totals = 0;
int sort_size = 0;
int sort_active = 0;
int set_debug = 0;
int show_ops = 0;
int show_activity = 0;

/* Debug options */
int sanity = 0;
int redzone = 0;
int poison = 0;
int tracking = 0;
int tracing = 0;

int page_size;

regex_t pattern;

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

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

static void usage(void)
{
    printf("slabinfo 5/7/2007. (c) 2007 sgi.\n\n"
        "slabinfo [-ahnpvtsz] [-d debugopts] [slab-regexp]\n"
        "-a|--aliases           Show aliases\n"
        "-A|--activity          Most active slabs first\n"
        "-d<options>|--debug=<options> Set/Clear Debug options\n"
        "-D|--display-active    Switch line format to activity\n"
        "-e|--empty             Show empty slabs\n"
        "-f|--first-alias       Show first alias\n"
        "-h|--help              Show usage information\n"
        "-i|--inverted          Inverted list\n"
        "-l|--slabs             Show slabs\n"
        "-n|--numa              Show NUMA information\n"
        "-o|--ops       Show kmem_cache_ops\n"
        "-s|--shrink            Shrink slabs\n"
        "-r|--report        Detailed report on single slabs\n"
        "-S|--Size              Sort by size\n"
        "-t|--tracking          Show alloc/free information\n"
        "-T|--Totals            Show summary information\n"
        "-v|--validate          Validate slabs\n"
        "-z|--zero              Include empty slabs\n"
        "-1|--1ref              Single reference\n"
        "\nValid debug options (FZPUT may be combined)\n"
        "a / A          Switch on all debug options (=FZUP)\n"
        "-              Switch off all debug options\n"
        "f / F          Sanity Checks (SLAB_DEBUG_FREE)\n"
        "z / Z          Redzoning\n"
        "p / P          Poisoning\n"
        "u / U          Tracking\n"
        "t / T          Tracing\n"
    );
}
```

```
static unsigned long read_obj(const char *name)
{
    FILE *f = fopen(name, "r");

    if (!f)
        buffer[0] = 0;
    else {
        if (!fgets(buffer, sizeof(buffer), f))
            buffer[0] = 0;
        fclose(f);
        if (buffer[strlen(buffer)] == '\n')
            buffer[strlen(buffer)] = 0;
    }
    return strlen(buffer);
}


/*
 * Get the contents of an attribute
 */
static unsigned long get_obj(const char *name)
{
    if (!read_obj(name))
        return 0;

    return atol(buffer);
}

static unsigned long get_obj_and_str(const char *name, char **x)
{
    unsigned long result = 0;
    char *p;

    *x = NULL;

    if (!read_obj(name)) {
        x = NULL;
        return 0;
    }
    result = strtoul(buffer, &p, 10);
    while (*p == ' ')
        p++;
    if (*p)
        *x = strdup(p);
    return result;
}

static void set_obj(struct slabinfo *s, const char *name, int n)
{
    char x[100];
    FILE *f;

    snprintf(x, 100, "%s/%s", s->name, name);
    f = fopen(x, "w");
    if (!f)
        fatal("Cannot write to %s\n", x);

    fprintf(f, "%d\n", n);
    fclose(f);
}

static unsigned long read_slab_obj(struct slabinfo *s, const char *name)
{
    char x[100];
    FILE *f;
```

```
    size_t l;

    snprintf(x, 100, "%s/%s", s->name, name);
    f = fopen(x, "r");
    if (!f) {
        buffer[0] = 0;
        l = 0;
    } else {
        l = fread(buffer, 1, sizeof(buffer), f);
        buffer[l] = 0;
        fclose(f);
    }
    return l;
}


/*
 * Put a size string together
 */
static int store_size(char *buffer, unsigned long value)
{
    unsigned long divisor = 1;
    char trailer = 0;
    int n;

    if (value > 1000000000UL) {
        divisor = 100000000UL;
        trailer = 'G';
    } else if (value > 1000000UL) {
        divisor = 100000UL;
        trailer = 'M';
    } else if (value > 1000UL) {
        divisor = 100;
        trailer = 'K';
    }

    value /= divisor;
    n = sprintf(buffer, "%ld",value);
    if (trailer) {
        buffer[n] = trailer;
        n++;
        buffer[n] = 0;
    }
    if (divisor != 1) {
        memmove(buffer + n - 2, buffer + n - 3, 4);
        buffer[n-2] = '.';
        n++;
    }
    return n;
}

static void decode_numa_list(int *numa, char *t)
{
    int node;
    int nr;

    memset(numa, 0, MAX_NODES * sizeof(int));

    if (!t)
        return;

    while (*t == 'N') {
        t++;
        node = strtoul(t, &t, 10);
        if (*t == '=') {
            t++;
```

```
            nr = strtoul(t, &t, 10);
            numa[node] = nr;
            if (node > highest_node)
                highest_node = node;
        }
        while (*t == ' ')
            t++;
    }
}

static void slab_validate(struct slabinfo *s)
{
    if (strcmp(s->name, "*") == 0)
        return;

    set_obj(s, "validate", 1);
}

static void slab_shrink(struct slabinfo *s)
{
    if (strcmp(s->name, "*") == 0)
        return;

    set_obj(s, "shrink", 1);
}

int line = 0;

static void first_line(void)
{
    if (show_activity)
        printf("Name                   Objects      Alloc       Free   %%Fast Fallb O\n");
    else
        printf("Name                   Objects Objsize    Space "
            "Slabs/Part/Cpu  O/S O %%Fr %%Ef Flg\n");
}

/*
 * Find the shortest alias of a slab
 */
static struct aliasinfo *find_one_alias(struct slabinfo *find)
{
    struct aliasinfo *a;
    struct aliasinfo *best = NULL;

    for(a = aliasinfo;a < aliasinfo + aliases; a++) {
        if (a->slab == find &&
            (!best || strlen(best->name) < strlen(a->name))) {
                best = a;
                if (strncmp(a->name,"kmall", 5) == 0)
                    return best;
            }
    }
    return best;
}

static unsigned long slab_size(struct slabinfo *s)
{
    return  s->slabs * (page_size << s->order);
}

static unsigned long slab_activity(struct slabinfo *s)
{
    return  s->alloc_fastpath + s->free_fastpath +
        s->alloc_slowpath + s->free_slowpath;
}
```

```
static void slab_numa(struct slabinfo *s, int mode)
{
    int node;

    if (strcmp(s->name, "*") == 0)
        return;

    if (!highest_node) {
        printf("\n%s: No NUMA information available.\n", s->name);
        return;
    }

    if (skip_zero && !s->slabs)
        return;

    if (!line) {
        printf("\n%-21s:", mode ? "NUMA nodes" : "Slab");
        for(node = 0; node <= highest_node; node++)
            printf(" %4d", node);
        printf("\n----------------------");
        for(node = 0; node <= highest_node; node++)
            printf("-----");
        printf("\n");
    }
    printf("%-21s ", mode ? "All slabs" : s->name);
    for(node = 0; node <= highest_node; node++) {
        char b[20];

        store_size(b, s->numa[node]);
        printf(" %4s", b);
    }
    printf("\n");
    if (mode) {
        printf("%-21s ", "Partial slabs");
        for(node = 0; node <= highest_node; node++) {
            char b[20];

            store_size(b, s->numa_partial[node]);
            printf(" %4s", b);
        }
        printf("\n");
    }
    line++;
}

static void show_tracking(struct slabinfo *s)
{
    printf("\n%s: Kernel object allocation\n", s->name);
    printf("-----------------------------------------------------------------------\n");
    if (read_slab_obj(s, "alloc_calls"))
        printf(buffer);
    else
        printf("No Data\n");

    printf("\n%s: Kernel object freeing\n", s->name);
    printf("-----------------------------------------------------------------------\n");
    if (read_slab_obj(s, "free_calls"))
        printf(buffer);
    else
        printf("No Data\n");

}

static void ops(struct slabinfo *s)
{
```

```
    if (strcmp(s->name, "*") == 0)
        return;

    if (read_slab_obj(s, "ops")) {
        printf("\n%s: kmem_cache operations\n", s->name);
        printf("--------------------------------------------\n");
        printf(buffer);
    } else
        printf("\n%s has no kmem_cache operations\n", s->name);
}

static const char *onoff(int x)
{
    if (x)
        return "On ";
    return "Off";
}

static void slab_stats(struct slabinfo *s)
{
    unsigned long total_alloc;
    unsigned long total_free;
    unsigned long total;

    if (!s->alloc_slab)
        return;

    total_alloc = s->alloc_fastpath + s->alloc_slowpath;
    total_free = s->free_fastpath + s->free_slowpath;

    if (!total_alloc)
        return;

    printf("\n");
    printf("Slab Perf Counter       Alloc     Free %%Al %%Fr\n");
    printf("--------------------------------------------------\n");
    printf("Fastpath             %8lu %8lu %3lu %3lu\n",
        s->alloc_fastpath, s->free_fastpath,
        s->alloc_fastpath * 100 / total_alloc,
        s->free_fastpath * 100 / total_free);
    printf("Slowpath             %8lu %8lu %3lu %3lu\n",
        total_alloc - s->alloc_fastpath, s->free_slowpath,
        (total_alloc - s->alloc_fastpath) * 100 / total_alloc,
        s->free_slowpath * 100 / total_free);
    printf("Page Alloc           %8lu %8lu %3lu %3lu\n",
        s->alloc_slab, s->free_slab,
        s->alloc_slab * 100 / total_alloc,
        s->free_slab * 100 / total_free);
    printf("Add partial          %8lu %8lu %3lu %3lu\n",
        s->deactivate_to_head + s->deactivate_to_tail,
        s->free_add_partial,
        (s->deactivate_to_head + s->deactivate_to_tail) * 100 / total_alloc,
        s->free_add_partial * 100 / total_free);
    printf("Remove partial       %8lu %8lu %3lu %3lu\n",
        s->alloc_from_partial, s->free_remove_partial,
        s->alloc_from_partial * 100 / total_alloc,
        s->free_remove_partial * 100 / total_free);

    printf("RemoteObj/SlabFrozen %8lu %8lu %3lu %3lu\n",
        s->deactivate_remote_frees, s->free_frozen,
        s->deactivate_remote_frees * 100 / total_alloc,
        s->free_frozen * 100 / total_free);

    printf("Total                %8lu %8lu\n\n", total_alloc, total_free);

    if (s->cpuslab_flush)
```

```
        printf("Flushes %8lu\n", s->cpuslab_flush);

    if (s->alloc_refill)
        printf("Refill %8lu\n", s->alloc_refill);

    total = s->deactivate_full + s->deactivate_empty +
            s->deactivate_to_head + s->deactivate_to_tail;

    if (total)
        printf("Deactivate Full=%lu(%lu%%) Empty=%lu(%lu%%) "
            "ToHead=%lu(%lu%%) ToTail=%lu(%lu%%)\n",
            s->deactivate_full, (s->deactivate_full * 100) / total,
            s->deactivate_empty, (s->deactivate_empty * 100) / total,
            s->deactivate_to_head, (s->deactivate_to_head * 100) / total,
            s->deactivate_to_tail, (s->deactivate_to_tail * 100) / total);
}

static void report(struct slabinfo *s)
{
    if (strcmp(s->name, "*") == 0)
        return;

    printf("\nSlabcache: %-20s  Aliases: %2d Order : %2d Objects: %lu\n",
        s->name, s->aliases, s->order, s->objects);
    if (s->hwcache_align)
        printf("** Hardware cacheline aligned\n");
    if (s->cache_dma)
        printf("** Memory is allocated in a special DMA zone\n");
    if (s->destroy_by_rcu)
        printf("** Slabs are destroyed via RCU\n");
    if (s->reclaim_account)
        printf("** Reclaim accounting active\n");

    printf("\nSizes (bytes)     Slabs              Debug                Memory\n");
    printf("------------------------------------------------------------------------\n");
    printf("Object : %7d  Total  : %7ld   Sanity Checks : %s  Total: %7ld\n",
            s->object_size, s->slabs, onoff(s->sanity_checks),
            s->slabs * (page_size << s->order));
    printf("SlabObj: %7d  Full   : %7ld   Redzoning     : %s  Used : %7ld\n",
            s->slab_size, s->slabs - s->partial - s->cpu_slabs,
            onoff(s->red_zone), s->objects * s->object_size);
    printf("SlabSiz: %7d  Partial: %7ld   Poisoning     : %s  Loss : %7ld\n",
            page_size << s->order, s->partial, onoff(s->poison),
            s->slabs * (page_size << s->order) - s->objects * s->object_size);
    printf("Loss   : %7d  CpuSlab: %7d   Tracking      : %s  Lalig: %7ld\n",
            s->slab_size - s->object_size, s->cpu_slabs, onoff(s->store_user),
            (s->slab_size - s->object_size) * s->objects);
    printf("Align  : %7d  Objects: %7d   Tracing       : %s  Lpadd: %7ld\n",
            s->align, s->objs_per_slab, onoff(s->trace),
            ((page_size << s->order) - s->objs_per_slab * s->slab_size) *
            s->slabs);

    ops(s);
    show_tracking(s);
    slab_numa(s, 1);
    slab_stats(s);
}

static void slabcache(struct slabinfo *s)
{
    char size_str[20];
    char dist_str[40];
    char flags[20];
    char *p = flags;

    if (strcmp(s->name, "*") == 0)
```

```
        return;

    if (actual_slabs == 1) {
        report(s);
        return;
    }

    if (skip_zero && !show_empty && !s->slabs)
        return;

    if (show_empty && s->slabs)
        return;

    store_size(size_str, slab_size(s));
    snprintf(dist_str, 40, "%lu/%lu/%d", s->slabs - s->cpu_slabs,
                        s->partial, s->cpu_slabs);

    if (!line++)
        first_line();

    if (s->aliases)
        *p++ = '*';
    if (s->cache_dma)
        *p++ = 'd';
    if (s->hwcache_align)
        *p++ = 'A';
    if (s->poison)
        *p++ = 'P';
    if (s->reclaim_account)
        *p++ = 'a';
    if (s->red_zone)
        *p++ = 'Z';
    if (s->sanity_checks)
        *p++ = 'F';
    if (s->store_user)
        *p++ = 'U';
    if (s->trace)
        *p++ = 'T';

    *p = 0;
    if (show_activity) {
        unsigned long total_alloc;
        unsigned long total_free;

        total_alloc = s->alloc_fastpath + s->alloc_slowpath;
        total_free = s->free_fastpath + s->free_slowpath;

        printf("%-21s %8ld %10ld %10ld %3ld %3ld %5ld %1d\n",
            s->name, s->objects,
            total_alloc, total_free,
            total_alloc ? (s->alloc_fastpath * 100 / total_alloc) : 0,
            total_free ? (s->free_fastpath * 100 / total_free) : 0,
            s->order_fallback, s->order);
    }
    else
        printf("%-21s %8ld %7d %8s %14s %4d %1d %3ld %3ld %s\n",
            s->name, s->objects, s->object_size, size_str, dist_str,
            s->objs_per_slab, s->order,
            s->slabs ? (s->partial * 100) / s->slabs : 100,
            s->slabs ? (s->objects * s->object_size * 100) /
                (s->slabs * (page_size << s->order)) : 100,
            flags);
}

/*
 * Analyze debug options. Return false if something is amiss.
```

```
 */
static int debug_opt_scan(char *opt)
{
    if (!opt || !opt[0] || strcmp(opt, "-") == 0)
        return 1;

    if (strcasecmp(opt, "a") == 0) {
        sanity = 1;
        poison = 1;
        redzone = 1;
        tracking = 1;
        return 1;
    }

    for ( ; *opt; opt++)
        switch (*opt) {
        case 'F' : case 'f':
            if (sanity)
                return 0;
            sanity = 1;
            break;
        case 'P' : case 'p':
            if (poison)
                return 0;
            poison = 1;
            break;

        case 'Z' : case 'z':
            if (redzone)
                return 0;
            redzone = 1;
            break;

        case 'U' : case 'u':
            if (tracking)
                return 0;
            tracking = 1;
            break;

        case 'T' : case 't':
            if (tracing)
                return 0;
            tracing = 1;
            break;
        default:
            return 0;
        }
    return 1;
}

static int slab_empty(struct slabinfo *s)
{
    if (s->objects > 0)
        return 0;

    /*
     * We may still have slabs even if there are no objects. Shrinking will
     * remove them.
     */
    if (s->slabs != 0)
        set_obj(s, "shrink", 1);

    return 1;
}

static void slab_debug(struct slabinfo *s)
```

```
{
    if (strcmp(s->name, "*") == 0)
        return;

    if (sanity && !s->sanity_checks) {
        set_obj(s, "sanity", 1);
    }
    if (!sanity && s->sanity_checks) {
        if (slab_empty(s))
            set_obj(s, "sanity", 0);
        else
            fprintf(stderr, "%s not empty cannot disable sanity checks\n", s->name);
    }
    if (redzone && !s->red_zone) {
        if (slab_empty(s))
            set_obj(s, "red_zone", 1);
        else
            fprintf(stderr, "%s not empty cannot enable redzoning\n", s->name);
    }
    if (!redzone && s->red_zone) {
        if (slab_empty(s))
            set_obj(s, "red_zone", 0);
        else
            fprintf(stderr, "%s not empty cannot disable redzoning\n", s->name);
    }
    if (poison && !s->poison) {
        if (slab_empty(s))
            set_obj(s, "poison", 1);
        else
            fprintf(stderr, "%s not empty cannot enable poisoning\n", s->name);
    }
    if (!poison && s->poison) {
        if (slab_empty(s))
            set_obj(s, "poison", 0);
        else
            fprintf(stderr, "%s not empty cannot disable poisoning\n", s->name);
    }
    if (tracking && !s->store_user) {
        if (slab_empty(s))
            set_obj(s, "store_user", 1);
        else
            fprintf(stderr, "%s not empty cannot enable tracking\n", s->name);
    }
    if (!tracking && s->store_user) {
        if (slab_empty(s))
            set_obj(s, "store_user", 0);
        else
            fprintf(stderr, "%s not empty cannot disable tracking\n", s->name);
    }
    if (tracing && !s->trace) {
        if (slabs == 1)
            set_obj(s, "trace", 1);
        else
            fprintf(stderr, "%s can only enable trace for one slab at a time\n", s->name);
    }
    if (!tracing && s->trace)
        set_obj(s, "trace", 1);
}

static void totals(void)
{
    struct slabinfo *s;

    int used_slabs = 0;
    char b1[20], b2[20], b3[20], b4[20];
    unsigned long long max = 1ULL << 63;
```

```
/* Object size */
unsigned long long min_objsize = max, max_objsize = 0, avg_objsize;

/* Number of partial slabs in a slabcache */
unsigned long long min_partial = max, max_partial = 0,
            avg_partial, total_partial = 0;

/* Number of slabs in a slab cache */
unsigned long long min_slabs = max, max_slabs = 0,
            avg_slabs, total_slabs = 0;

/* Size of the whole slab */
unsigned long long min_size = max, max_size = 0,
            avg_size, total_size = 0;

/* Bytes used for object storage in a slab */
unsigned long long min_used = max, max_used = 0,
            avg_used, total_used = 0;

/* Waste: Bytes used for alignment and padding */
unsigned long long min_waste = max, max_waste = 0,
            avg_waste, total_waste = 0;
/* Number of objects in a slab */
unsigned long long min_objects = max, max_objects = 0,
            avg_objects, total_objects = 0;
/* Waste per object */
unsigned long long min_objwaste = max,
            max_objwaste = 0, avg_objwaste,
            total_objwaste = 0;

/* Memory per object */
unsigned long long min_memobj = max,
            max_memobj = 0, avg_memobj,
            total_objsize = 0;

/* Percentage of partial slabs per slab */
unsigned long min_ppart = 100, max_ppart = 0,
            avg_ppart, total_ppart = 0;

/* Number of objects in partial slabs */
unsigned long min_partobj = max, max_partobj = 0,
            avg_partobj, total_partobj = 0;

/* Percentage of partial objects of all objects in a slab */
unsigned long min_ppartobj = 100, max_ppartobj = 0,
            avg_ppartobj, total_ppartobj = 0;


for (s = slabinfo; s < slabinfo + slabs; s++) {
    unsigned long long size;
    unsigned long used;
    unsigned long long wasted;
    unsigned long long objwaste;
    unsigned long percentage_partial_slabs;
    unsigned long percentage_partial_objs;

    if (!s->slabs || !s->objects)
        continue;

    used_slabs++;

    size = slab_size(s);
    used = s->objects * s->object_size;
    wasted = size - used;
    objwaste = s->slab_size - s->object_size;
```

```
percentage_partial_slabs = s->partial * 100 / s->slabs;
if (percentage_partial_slabs > 100)
    percentage_partial_slabs = 100;

percentage_partial_objs = s->objects_partial * 100
                    / s->objects;

if (percentage_partial_objs > 100)
    percentage_partial_objs = 100;

if (s->object_size < min_objsize)
    min_objsize = s->object_size;
if (s->partial < min_partial)
    min_partial = s->partial;
if (s->slabs < min_slabs)
    min_slabs = s->slabs;
if (size < min_size)
    min_size = size;
if (wasted < min_waste)
    min_waste = wasted;
if (objwaste < min_objwaste)
    min_objwaste = objwaste;
if (s->objects < min_objects)
    min_objects = s->objects;
if (used < min_used)
    min_used = used;
if (s->objects_partial < min_partobj)
    min_partobj = s->objects_partial;
if (percentage_partial_slabs < min_ppart)
    min_ppart = percentage_partial_slabs;
if (percentage_partial_objs < min_ppartobj)
    min_ppartobj = percentage_partial_objs;
if (s->slab_size < min_memobj)
    min_memobj = s->slab_size;

if (s->object_size > max_objsize)
    max_objsize = s->object_size;
if (s->partial > max_partial)
    max_partial = s->partial;
if (s->slabs > max_slabs)
    max_slabs = s->slabs;
if (size > max_size)
    max_size = size;
if (wasted > max_waste)
    max_waste = wasted;
if (objwaste > max_objwaste)
    max_objwaste = objwaste;
if (s->objects > max_objects)
    max_objects = s->objects;
if (used > max_used)
    max_used = used;
if (s->objects_partial > max_partobj)
    max_partobj = s->objects_partial;
if (percentage_partial_slabs > max_ppart)
    max_ppart = percentage_partial_slabs;
if (percentage_partial_objs > max_ppartobj)
    max_ppartobj = percentage_partial_objs;
if (s->slab_size > max_memobj)
    max_memobj = s->slab_size;

total_partial += s->partial;
total_slabs += s->slabs;
total_size += size;
total_waste += wasted;
```

```
    total_objects += s->objects;
    total_used += used;
    total_partobj += s->objects_partial;
    total_ppart += percentage_partial_slabs;
    total_ppartobj += percentage_partial_objs;

    total_objwaste += s->objects * objwaste;
    total_objsize += s->objects * s->slab_size;
}

if (!total_objects) {
    printf("No objects\n");
    return;
}
if (!used_slabs) {
    printf("No slabs\n");
    return;
}

/* Per slab averages */
avg_partial = total_partial / used_slabs;
avg_slabs = total_slabs / used_slabs;
avg_size = total_size / used_slabs;
avg_waste = total_waste / used_slabs;

avg_objects = total_objects / used_slabs;
avg_used = total_used / used_slabs;
avg_partobj = total_partobj / used_slabs;
avg_ppart = total_ppart / used_slabs;
avg_ppartobj = total_ppartobj / used_slabs;

/* Per object object sizes */
avg_objsize = total_used / total_objects;
avg_objwaste = total_objwaste / total_objects;
avg_partobj = total_partobj * 100 / total_objects;
avg_memobj = total_objsize / total_objects;

printf("Slabcache Totals\n");
printf("----------------\n");
printf("Slabcaches : %3d      Aliases  : %3d->%-3d Active: %3d\n",
        slabs, aliases, alias_targets, used_slabs);

store_size(b1, total_size);store_size(b2, total_waste);
store_size(b3, total_waste * 100 / total_used);
printf("Memory used: %6s   # Loss   : %6s   MRatio:%6s%%\n", b1, b2, b3);

store_size(b1, total_objects);store_size(b2, total_partobj);
store_size(b3, total_partobj * 100 / total_objects);
printf("# Objects  : %6s   # PartObj: %6s   ORatio:%6s%%\n", b1, b2, b3);

printf("\n");
printf("Per Cache    Average         Min         Max       Total\n");
printf("---------------------------------------------------------\n");

store_size(b1, avg_objects);store_size(b2, min_objects);
store_size(b3, max_objects);store_size(b4, total_objects);
printf("#Objects  %10s  %10s  %10s  %10s\n",
        b1, b2, b3, b4);

store_size(b1, avg_slabs);store_size(b2, min_slabs);
store_size(b3, max_slabs);store_size(b4, total_slabs);
printf("#Slabs    %10s  %10s  %10s  %10s\n",
        b1, b2, b3, b4);

store_size(b1, avg_partial);store_size(b2, min_partial);
store_size(b3, max_partial);store_size(b4, total_partial);
```

```
    printf("#PartSlab %10s  %10s  %10s  %10s\n",
            b1, b2, b3, b4);
    store_size(b1, avg_ppart);store_size(b2, min_ppart);
    store_size(b3, max_ppart);
    store_size(b4, total_partial * 100  / total_slabs);
    printf("%%PartSlab%10s%% %10s%% %10s%% %10s%%\n",
            b1, b2, b3, b4);

    store_size(b1, avg_partobj);store_size(b2, min_partobj);
    store_size(b3, max_partobj);
    store_size(b4, total_partobj);
    printf("PartObjs  %10s  %10s  %10s  %10s\n",
            b1, b2, b3, b4);

    store_size(b1, avg_ppartobj);store_size(b2, min_ppartobj);
    store_size(b3, max_ppartobj);
    store_size(b4, total_partobj * 100 / total_objects);
    printf("%% PartObj%10s%% %10s%% %10s%% %10s%%\n",
            b1, b2, b3, b4);

    store_size(b1, avg_size);store_size(b2, min_size);
    store_size(b3, max_size);store_size(b4, total_size);
    printf("Memory    %10s  %10s  %10s  %10s\n",
            b1, b2, b3, b4);

    store_size(b1, avg_used);store_size(b2, min_used);
    store_size(b3, max_used);store_size(b4, total_used);
    printf("Used      %10s  %10s  %10s  %10s\n",
            b1, b2, b3, b4);

    store_size(b1, avg_waste);store_size(b2, min_waste);
    store_size(b3, max_waste);store_size(b4, total_waste);
    printf("Loss      %10s  %10s  %10s  %10s\n",
            b1, b2, b3, b4);

    printf("\n");
    printf("Per Object   Average         Min         Max\n");
    printf("---------------------------------------------\n");

    store_size(b1, avg_memobj);store_size(b2, min_memobj);
    store_size(b3, max_memobj);
    printf("Memory    %10s  %10s  %10s\n",
            b1, b2, b3);
    store_size(b1, avg_objsize);store_size(b2, min_objsize);
    store_size(b3, max_objsize);
    printf("User      %10s  %10s  %10s\n",
            b1, b2, b3);

    store_size(b1, avg_objwaste);store_size(b2, min_objwaste);
    store_size(b3, max_objwaste);
    printf("Loss      %10s  %10s  %10s\n",
            b1, b2, b3);
}

static void sort_slabs(void)
{
    struct slabinfo *s1,*s2;

    for (s1 = slabinfo; s1 < slabinfo + slabs; s1++) {
        for (s2 = s1 + 1; s2 < slabinfo + slabs; s2++) {
            int result;

            if (sort_size)
                result = slab_size(s1) < slab_size(s2);
            else if (sort_active)
                result = slab_activity(s1) < slab_activity(s2);
```

```
            else
                result = strcasecmp(s1->name, s2->name);

            if (show_inverted)
                result = -result;

            if (result > 0) {
                struct slabinfo t;

                memcpy(&t, s1, sizeof(struct slabinfo));
                memcpy(s1, s2, sizeof(struct slabinfo));
                memcpy(s2, &t, sizeof(struct slabinfo));
            }
        }
    }
}

static void sort_aliases(void)
{
    struct aliasinfo *a1,*a2;

    for (a1 = aliasinfo; a1 < aliasinfo + aliases; a1++) {
        for (a2 = a1 + 1; a2 < aliasinfo + aliases; a2++) {
            char *n1, *n2;

            n1 = a1->name;
            n2 = a2->name;
            if (show_alias && !show_inverted) {
                n1 = a1->ref;
                n2 = a2->ref;
            }
            if (strcasecmp(n1, n2) > 0) {
                struct aliasinfo t;

                memcpy(&t, a1, sizeof(struct aliasinfo));
                memcpy(a1, a2, sizeof(struct aliasinfo));
                memcpy(a2, &t, sizeof(struct aliasinfo));
            }
        }
    }
}

static void link_slabs(void)
{
    struct aliasinfo *a;
    struct slabinfo *s;

    for (a = aliasinfo; a < aliasinfo + aliases; a++) {

        for (s = slabinfo; s < slabinfo + slabs; s++)
            if (strcmp(a->ref, s->name) == 0) {
                a->slab = s;
                s->refs++;
                break;
            }
        if (s == slabinfo + slabs)
            fatal("Unresolved alias %s\n", a->ref);
    }
}

static void alias(void)
{
    struct aliasinfo *a;
    char *active = NULL;

    sort_aliases();
```

```
    link_slabs();

    for(a = aliasinfo; a < aliasinfo + aliases; a++) {

        if (!show_single_ref && a->slab->refs == 1)
            continue;

        if (!show_inverted) {
            if (active) {
                if (strcmp(a->slab->name, active) == 0) {
                    printf(" %s", a->name);
                    continue;
                }
            }
            printf("\n%-12s <- %s", a->slab->name, a->name);
            active = a->slab->name;
        }
        else
            printf("%-20s -> %s\n", a->name, a->slab->name);
    }
    if (active)
        printf("\n");
}


static void rename_slabs(void)
{
    struct slabinfo *s;
    struct aliasinfo *a;

    for (s = slabinfo; s < slabinfo + slabs; s++) {
        if (*s->name != ':')
            continue;

        if (s->refs > 1 && !show_first_alias)
            continue;

        a = find_one_alias(s);

        if (a)
            s->name = a->name;
        else {
            s->name = "*";
            actual_slabs--;
        }
    }
}

static int slab_mismatch(char *slab)
{
    return regexec(&pattern, slab, 0, NULL, 0);
}

static void read_slab_dir(void)
{
    DIR *dir;
    struct dirent *de;
    struct slabinfo *slab = slabinfo;
    struct aliasinfo *alias = aliasinfo;
    char *p;
    char *t;
    int count;

    if (chdir("/sys/kernel/slab") && chdir("/sys/slab"))
        fatal("SYSFS support for SLUB not active\n");
```

```
dir = opendir(".");
while ((de = readdir(dir))) {
    if (de->d_name[0] == '.' ||
        (de->d_name[0] != ':' && slab_mismatch(de->d_name)))
            continue;
    switch (de->d_type) {
       case DT_LNK:
        alias->name = strdup(de->d_name);
        count = readlink(de->d_name, buffer, sizeof(buffer));

        if (count < 0)
            fatal("Cannot read symlink %s\n", de->d_name);

        buffer[count] = 0;
        p = buffer + count;
        while (p > buffer && p[-1] != '/')
            p--;
        alias->ref = strdup(p);
        alias++;
        break;
       case DT_DIR:
        if (chdir(de->d_name))
            fatal("Unable to access slab %s\n", slab->name);
        slab->name = strdup(de->d_name);
        slab->alias = 0;
        slab->refs = 0;
        slab->aliases = get_obj("aliases");
        slab->align = get_obj("align");
        slab->cache_dma = get_obj("cache_dma");
        slab->cpu_slabs = get_obj("cpu_slabs");
        slab->destroy_by_rcu = get_obj("destroy_by_rcu");
        slab->hwcache_align = get_obj("hwcache_align");
        slab->object_size = get_obj("object_size");
        slab->objects = get_obj("objects");
        slab->objects_partial = get_obj("objects_partial");
        slab->objects_total = get_obj("objects_total");
        slab->objs_per_slab = get_obj("objs_per_slab");
        slab->order = get_obj("order");
        slab->partial = get_obj("partial");
        slab->partial = get_obj_and_str("partial", &t);
        decode_numa_list(slab->numa_partial, t);
        free(t);
        slab->poison = get_obj("poison");
        slab->reclaim_account = get_obj("reclaim_account");
        slab->red_zone = get_obj("red_zone");
        slab->sanity_checks = get_obj("sanity_checks");
        slab->slab_size = get_obj("slab_size");
        slab->slabs = get_obj_and_str("slabs", &t);
        decode_numa_list(slab->numa, t);
        free(t);
        slab->store_user = get_obj("store_user");
        slab->trace = get_obj("trace");
        slab->alloc_fastpath = get_obj("alloc_fastpath");
        slab->alloc_slowpath = get_obj("alloc_slowpath");
        slab->free_fastpath = get_obj("free_fastpath");
        slab->free_slowpath = get_obj("free_slowpath");
        slab->free_frozen= get_obj("free_frozen");
        slab->free_add_partial = get_obj("free_add_partial");
        slab->free_remove_partial = get_obj("free_remove_partial");
        slab->alloc_from_partial = get_obj("alloc_from_partial");
        slab->alloc_slab = get_obj("alloc_slab");
        slab->alloc_refill = get_obj("alloc_refill");
        slab->free_slab = get_obj("free_slab");
        slab->cpuslab_flush = get_obj("cpuslab_flush");
        slab->deactivate_full = get_obj("deactivate_full");
        slab->deactivate_empty = get_obj("deactivate_empty");
```

```
            slab->deactivate_to_head = get_obj("deactivate_to_head");
            slab->deactivate_to_tail = get_obj("deactivate_to_tail");
            slab->deactivate_remote_frees = get_obj("deactivate_remote_frees");
            slab->order_fallback = get_obj("order_fallback");
            chdir("..");
            if (slab->name[0] == ':')
                alias_targets++;
            slab++;
            break;
           default :
            fatal("Unknown file type %lx\n", de->d_type);
        }
    }
    closedir(dir);
    slabs = slab - slabinfo;
    actual_slabs = slabs;
    aliases = alias - aliasinfo;
    if (slabs > MAX_SLABS)
        fatal("Too many slabs\n");
    if (aliases > MAX_ALIASES)
        fatal("Too many aliases\n");
}

static void output_slabs(void)
{
    struct slabinfo *slab;

    for (slab = slabinfo; slab < slabinfo + slabs; slab++) {

        if (slab->alias)
            continue;


        if (show_numa)
            slab_numa(slab, 0);
        else if (show_track)
            show_tracking(slab);
        else if (validate)
            slab_validate(slab);
        else if (shrink)
            slab_shrink(slab);
        else if (set_debug)
            slab_debug(slab);
        else if (show_ops)
            ops(slab);
        else if (show_slab)
            slabcache(slab);
        else if (show_report)
            report(slab);
    }
}

struct option opts[] = {
    { "aliases", 0, NULL, 'a' },
    { "activity", 0, NULL, 'A' },
    { "debug", 2, NULL, 'd' },
    { "display-activity", 0, NULL, 'D' },
    { "empty", 0, NULL, 'e' },
    { "first-alias", 0, NULL, 'f' },
    { "help", 0, NULL, 'h' },
    { "inverted", 0, NULL, 'i'},
    { "numa", 0, NULL, 'n' },
    { "ops", 0, NULL, 'o' },
    { "report", 0, NULL, 'r' },
    { "shrink", 0, NULL, 's' },
    { "slabs", 0, NULL, 'l' },
```

```
    { "track", 0, NULL, 't'},
    { "validate", 0, NULL, 'v' },
    { "zero", 0, NULL, 'z' },
    { "1ref", 0, NULL, '1'},
    { NULL, 0, NULL, 0 }
};

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

    page_size = getpagesize();

    while ((c = getopt_long(argc, argv, "aAd::Defhil1noprstvzTS",
                        opts, NULL)) != -1)
        switch (c) {
        case '1':
            show_single_ref = 1;
            break;
        case 'a':
            show_alias = 1;
            break;
        case 'A':
            sort_active = 1;
            break;
        case 'd':
            set_debug = 1;
            if (!debug_opt_scan(optarg))
                fatal("Invalid debug option '%s'\n", optarg);
            break;
        case 'D':
            show_activity = 1;
            break;
        case 'e':
            show_empty = 1;
            break;
        case 'f':
            show_first_alias = 1;
            break;
        case 'h':
            usage();
            return 0;
        case 'i':
            show_inverted = 1;
            break;
        case 'n':
            show_numa = 1;
            break;
        case 'o':
            show_ops = 1;
            break;
        case 'r':
            show_report = 1;
            break;
        case 's':
            shrink = 1;
            break;
        case 'l':
            show_slab = 1;
            break;
        case 't':
            show_track = 1;
            break;
        case 'v':
```

```
            validate = 1;
            break;
        case 'z':
            skip_zero = 0;
            break;
        case 'T':
            show_totals = 1;
            break;
        case 'S':
            sort_size = 1;
            break;

        default:
            fatal("%s: Invalid option '%c'\n", argv[0], optopt);

    }

    if (!show_slab && !show_alias && !show_track && !show_report
        && !validate && !shrink && !set_debug && !show_ops)
            show_slab = 1;

    if (argc > optind)
        pattern_source = argv[optind];
    else
        pattern_source = ".*";

    err = regcomp(&pattern, pattern_source, REG_ICASE|REG_NOSUB);
    if (err)
        fatal("%s: Invalid pattern '%s' code %d\n",
            argv[0], pattern_source, err);
    read_slab_dir();
    if (show_alias)
        alias();
    else
    if (show_totals)
        totals();
    else {
        link_slabs();
        rename_slabs();
        sort_slabs();
        output_slabs();
    }
    return 0;
}
```