/\*

\* lib/plist.c

\* \* Descending-priority-sorted double-linked list

\* \* (C) 2002-2003 Intel Corp

\* Inaky Perez-Gonzalez <inaky.perez-gonzalez@intel.com>.

\*

\* 2001-2005 (c) MontaVista Software, Inc.

\* Daniel Walker <dwalker@mvista.com>

\*

\* (C) 2005 Thomas Gleixner <tglx@linutronix.de>

\*

\* Simplifications of the original code by

\* Oleg Nesterov <oleg@tv-sign.ru>

\*

\* Licensed under the FSF's GNU Public License v2 or later.

\*

\* Based on simple lists (include/linux/list.h).

\*

\* This file contains the add / del functions which are considered to

\* be too large to inline. See include/linux/plist.h for further

\* information.

\*/

#include <linux/bug.h>

#include <linux/plist.h>

#ifdef CONFIG\_DEBUG\_PI\_LIST

static struct plist\_head test\_head;

static void plist\_check\_prev\_next(struct list\_head \*t, struct list\_head \*p,

struct list\_head \*n)

{

WARN(n->prev != p || p->next != n,

"top: %p, n: %p, p: %p\n"

"prev: %p, n: %p, p: %p\n"

"next: %p, n: %p, p: %p\n",

t, t->next, t->prev,

p, p->next, p->prev,

n, n->next, n->prev);

}

static void plist\_check\_list(struct list\_head \*top)

{

struct list\_head \*prev = top, \*next = top->next;

plist\_check\_prev\_next(top, prev, next);

while (next != top) {

prev = next;

next = prev->next;

plist\_check\_prev\_next(top, prev, next);

}

}

static void plist\_check\_head(struct plist\_head \*head)

{

if (!plist\_head\_empty(head))

plist\_check\_list(&plist\_first(head)->prio\_list);

plist\_check\_list(&head->node\_list);

}

#else

# define plist\_check\_head(h) do { } while (0)

#endif

/\*\*

\* plist\_add - add @node to @head

\*

\* @node: &struct plist\_node pointer

\* @head: &struct plist\_head pointer

\*/

void plist\_add(struct plist\_node \*node, struct plist\_head \*head)

{

struct plist\_node \*first, \*iter, \*prev = NULL;

struct list\_head \*node\_next = &head->node\_list;

plist\_check\_head(head);

WARN\_ON(!plist\_node\_empty(node));

WARN\_ON(!list\_empty(&node->prio\_list));

if (plist\_head\_empty(head))

goto ins\_node;

first = iter = plist\_first(head);

do {

if (node->prio < iter->prio) {

node\_next = &iter->node\_list;

break;

}

prev = iter;

iter = list\_entry(iter->prio\_list.next,

struct plist\_node, prio\_list);

} while (iter != first);

if (!prev || prev->prio != node->prio)

list\_add\_tail(&node->prio\_list, &iter->prio\_list);

ins\_node:

list\_add\_tail(&node->node\_list, node\_next);

plist\_check\_head(head);

}

/\*\*

\* plist\_del - Remove a @node from plist.

\*

\* @node: &struct plist\_node pointer - entry to be removed

\* @head: &struct plist\_head pointer - list head

\*/

void plist\_del(struct plist\_node \*node, struct plist\_head \*head)

{

plist\_check\_head(head);

if (!list\_empty(&node->prio\_list)) {

if (node->node\_list.next != &head->node\_list) {

struct plist\_node \*next;

next = list\_entry(node->node\_list.next,

struct plist\_node, node\_list);

/\* add the next plist\_node into prio\_list \*/

if (list\_empty(&next->prio\_list))

list\_add(&next->prio\_list, &node->prio\_list);

}

list\_del\_init(&node->prio\_list);

}

list\_del\_init(&node->node\_list);

plist\_check\_head(head);

}

/\*\*

\* plist\_requeue - Requeue @node at end of same-prio entries.

\*

\* This is essentially an optimized plist\_del() followed by

\* plist\_add(). It moves an entry already in the plist to

\* after any other same-priority entries.

\*

\* @node: &struct plist\_node pointer - entry to be moved

\* @head: &struct plist\_head pointer - list head

\*/

void plist\_requeue(struct plist\_node \*node, struct plist\_head \*head)

{

struct plist\_node \*iter;

struct list\_head \*node\_next = &head->node\_list;

plist\_check\_head(head);

BUG\_ON(plist\_head\_empty(head));

BUG\_ON(plist\_node\_empty(node));

if (node == plist\_last(head))

return;

iter = plist\_next(node);

if (node->prio != iter->prio)

return;

plist\_del(node, head);

plist\_for\_each\_continue(iter, head) {

if (node->prio != iter->prio) {

node\_next = &iter->node\_list;

break;

}

}

list\_add\_tail(&node->node\_list, node\_next);

plist\_check\_head(head);

}

#ifdef CONFIG\_DEBUG\_PI\_LIST

#include <linux/sched.h>

#include <linux/module.h>

#include <linux/init.h>

static struct plist\_node \_\_initdata test\_node[241];

static void \_\_init plist\_test\_check(int nr\_expect)

{

struct plist\_node \*first, \*prio\_pos, \*node\_pos;

if (plist\_head\_empty(&test\_head)) {

BUG\_ON(nr\_expect != 0);

return;

}

prio\_pos = first = plist\_first(&test\_head);

plist\_for\_each(node\_pos, &test\_head) {

if (nr\_expect-- < 0)

break;

if (node\_pos == first)

continue;

if (node\_pos->prio == prio\_pos->prio) {

BUG\_ON(!list\_empty(&node\_pos->prio\_list));

continue;

}

BUG\_ON(prio\_pos->prio > node\_pos->prio);

BUG\_ON(prio\_pos->prio\_list.next != &node\_pos->prio\_list);

prio\_pos = node\_pos;

}

BUG\_ON(nr\_expect != 0);

BUG\_ON(prio\_pos->prio\_list.next != &first->prio\_list);

}

static void \_\_init plist\_test\_requeue(struct plist\_node \*node)

{

plist\_requeue(node, &test\_head);

if (node != plist\_last(&test\_head))

BUG\_ON(node->prio == plist\_next(node)->prio);

}

static int \_\_init plist\_test(void)

{

int nr\_expect = 0, i, loop;

unsigned int r = local\_clock();

printk(KERN\_DEBUG "start plist test\n");

plist\_head\_init(&test\_head);

for (i = 0; i < ARRAY\_SIZE(test\_node); i++)

plist\_node\_init(test\_node + i, 0);

for (loop = 0; loop < 1000; loop++) {

r = r \* 193939 % 47629;

i = r % ARRAY\_SIZE(test\_node);

if (plist\_node\_empty(test\_node + i)) {

r = r \* 193939 % 47629;

test\_node[i].prio = r % 99;

plist\_add(test\_node + i, &test\_head);

nr\_expect++;

} else {

plist\_del(test\_node + i, &test\_head);

nr\_expect--;

}

plist\_test\_check(nr\_expect);

if (!plist\_node\_empty(test\_node + i)) {

plist\_test\_requeue(test\_node + i);

plist\_test\_check(nr\_expect);

}

}

for (i = 0; i < ARRAY\_SIZE(test\_node); i++) {

if (plist\_node\_empty(test\_node + i))

continue;

plist\_del(test\_node + i, &test\_head);

nr\_expect--;

plist\_test\_check(nr\_expect);

}

printk(KERN\_DEBUG "end plist test\n");

return 0;

}

module\_init(plist\_test);

#endif

/\*

\* Lock-less NULL terminated single linked list

\*

\* The basic atomic operation of this list is cmpxchg on long. On

\* architectures that don't have NMI-safe cmpxchg implementation, the

\* list can NOT be used in NMI handlers. So code that uses the list in

\* an NMI handler should depend on CONFIG\_ARCH\_HAVE\_NMI\_SAFE\_CMPXCHG.

\*

\* Copyright 2010,2011 Intel Corp.

\* Author: Huang Ying <ying.huang@intel.com>

\*

\* This program is free software; you can redistribute it and/or

\* modify it under the terms of the GNU General Public License version

\* 2 as published by the Free Software Foundation;

\*

\* 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 have received a copy of the GNU General Public License

\* along with this program; if not, write to the Free Software

\* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

\*/

#include <linux/kernel.h>

#include <linux/export.h>

#include <linux/llist.h>

/\*\*

\* llist\_add\_batch - add several linked entries in batch

\* @new\_first: first entry in batch to be added

\* @new\_last: last entry in batch to be added

\* @head: the head for your lock-less list

\*

\* Return whether list is empty before adding.

\*/

bool llist\_add\_batch(struct llist\_node \*new\_first, struct llist\_node \*new\_last,

struct llist\_head \*head)

{

struct llist\_node \*first;

do {

new\_last->next = first = ACCESS\_ONCE(head->first);

} while (cmpxchg(&head->first, first, new\_first) != first);

return !first;

}

EXPORT\_SYMBOL\_GPL(llist\_add\_batch);

/\*\*

\* llist\_del\_first - delete the first entry of lock-less list

\* @head: the head for your lock-less list

\*

\* If list is empty, return NULL, otherwise, return the first entry

\* deleted, this is the newest added one.

\*

\* Only one llist\_del\_first user can be used simultaneously with

\* multiple llist\_add users without lock. Because otherwise

\* llist\_del\_first, llist\_add, llist\_add (or llist\_del\_all, llist\_add,

\* llist\_add) sequence in another user may change @head->first->next,

\* but keep @head->first. If multiple consumers are needed, please

\* use llist\_del\_all or use lock between consumers.

\*/

struct llist\_node \*llist\_del\_first(struct llist\_head \*head)

{

struct llist\_node \*entry, \*old\_entry, \*next;

entry = smp\_load\_acquire(&head->first);

for (;;) {

if (entry == NULL)

return NULL;

old\_entry = entry;

next = READ\_ONCE(entry->next);

entry = cmpxchg(&head->first, old\_entry, next);

if (entry == old\_entry)

break;

}

return entry;

}

EXPORT\_SYMBOL\_GPL(llist\_del\_first);

/\*\*

\* llist\_reverse\_order - reverse order of a llist chain

\* @head: first item of the list to be reversed

\*

\* Reverse the order of a chain of llist entries and return the

\* new first entry.

\*/

struct llist\_node \*llist\_reverse\_order(struct llist\_node \*head)

{

struct llist\_node \*new\_head = NULL;

while (head) {

struct llist\_node \*tmp = head;

head = head->next;

tmp->next = new\_head;

new\_head = tmp;

}

return new\_head;

}

EXPORT\_SYMBOL\_GPL(llist\_reverse\_order);

/\*

\* Copyright (C) 2001 Momchil Velikov

\* Portions Copyright (C) 2001 Christoph Hellwig

\* Copyright (C) 2005 SGI, Christoph Lameter

\* Copyright (C) 2006 Nick Piggin

\* Copyright (C) 2012 Konstantin Khlebnikov

\*

\* 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; either version 2, or (at

\* your option) any later version.

\*

\* 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 have received a copy of the GNU General Public License

\* along with this program; if not, write to the Free Software

\* Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

\*/

#include <linux/errno.h>

#include <linux/init.h>

#include <linux/kernel.h>

#include <linux/export.h>

#include <linux/radix-tree.h>

#include <linux/percpu.h>

#include <linux/slab.h>

#include <linux/kmemleak.h>

#include <linux/notifier.h>

#include <linux/cpu.h>

#include <linux/string.h>

#include <linux/bitops.h>

#include <linux/rcupdate.h>

#include <linux/preempt.h> /\* in\_interrupt() \*/

/\*

\* The height\_to\_maxindex array needs to be one deeper than the maximum

\* path as height 0 holds only 1 entry.

\*/

static unsigned long height\_to\_maxindex[RADIX\_TREE\_MAX\_PATH + 1] \_\_read\_mostly;

/\*

\* Radix tree node cache.

\*/

static struct kmem\_cache \*radix\_tree\_node\_cachep;

/\*

\* The radix tree is variable-height, so an insert operation not only has

\* to build the branch to its corresponding item, it also has to build the

\* branch to existing items if the size has to be increased (by

\* radix\_tree\_extend).

\*

\* The worst case is a zero height tree with just a single item at index 0,

\* and then inserting an item at index ULONG\_MAX. This requires 2 new branches

\* of RADIX\_TREE\_MAX\_PATH size to be created, with only the root node shared.

\* Hence:

\*/

#define RADIX\_TREE\_PRELOAD\_SIZE (RADIX\_TREE\_MAX\_PATH \* 2 - 1)

/\*

\* Per-cpu pool of preloaded nodes

\*/

struct radix\_tree\_preload {

int nr;

/\* nodes->private\_data points to next preallocated node \*/

struct radix\_tree\_node \*nodes;

};

static DEFINE\_PER\_CPU(struct radix\_tree\_preload, radix\_tree\_preloads) = { 0, };

static inline void \*ptr\_to\_indirect(void \*ptr)

{

return (void \*)((unsigned long)ptr | RADIX\_TREE\_INDIRECT\_PTR);

}

static inline void \*indirect\_to\_ptr(void \*ptr)

{

return (void \*)((unsigned long)ptr & ~RADIX\_TREE\_INDIRECT\_PTR);

}

static inline gfp\_t root\_gfp\_mask(struct radix\_tree\_root \*root)

{

return root->gfp\_mask & \_\_GFP\_BITS\_MASK;

}

static inline void tag\_set(struct radix\_tree\_node \*node, unsigned int tag,

int offset)

{

\_\_set\_bit(offset, node->tags[tag]);

}

static inline void tag\_clear(struct radix\_tree\_node \*node, unsigned int tag,

int offset)

{

\_\_clear\_bit(offset, node->tags[tag]);

}

static inline int tag\_get(struct radix\_tree\_node \*node, unsigned int tag,

int offset)

{

return test\_bit(offset, node->tags[tag]);

}

static inline void root\_tag\_set(struct radix\_tree\_root \*root, unsigned int tag)

{

root->gfp\_mask |= (\_\_force gfp\_t)(1 << (tag + \_\_GFP\_BITS\_SHIFT));

}

static inline void root\_tag\_clear(struct radix\_tree\_root \*root, unsigned int tag)

{

root->gfp\_mask &= (\_\_force gfp\_t)~(1 << (tag + \_\_GFP\_BITS\_SHIFT));

}

static inline void root\_tag\_clear\_all(struct radix\_tree\_root \*root)

{

root->gfp\_mask &= \_\_GFP\_BITS\_MASK;

}

static inline int root\_tag\_get(struct radix\_tree\_root \*root, unsigned int tag)

{

return (\_\_force unsigned)root->gfp\_mask & (1 << (tag + \_\_GFP\_BITS\_SHIFT));

}

/\*

\* Returns 1 if any slot in the node has this tag set.

\* Otherwise returns 0.

\*/

static inline int any\_tag\_set(struct radix\_tree\_node \*node, unsigned int tag)

{

int idx;

for (idx = 0; idx < RADIX\_TREE\_TAG\_LONGS; idx++) {

if (node->tags[tag][idx])

return 1;

}

return 0;

}

/\*\*

\* radix\_tree\_find\_next\_bit - find the next set bit in a memory region

\*

\* @addr: The address to base the search on

\* @size: The bitmap size in bits

\* @offset: The bitnumber to start searching at

\*

\* Unrollable variant of find\_next\_bit() for constant size arrays.

\* Tail bits starting from size to roundup(size, BITS\_PER\_LONG) must be zero.

\* Returns next bit offset, or size if nothing found.

\*/

static \_\_always\_inline unsigned long

radix\_tree\_find\_next\_bit(const unsigned long \*addr,

unsigned long size, unsigned long offset)

{

if (!\_\_builtin\_constant\_p(size))

return find\_next\_bit(addr, size, offset);

if (offset < size) {

unsigned long tmp;

addr += offset / BITS\_PER\_LONG;

tmp = \*addr >> (offset % BITS\_PER\_LONG);

if (tmp)

return \_\_ffs(tmp) + offset;

offset = (offset + BITS\_PER\_LONG) & ~(BITS\_PER\_LONG - 1);

while (offset < size) {

tmp = \*++addr;

if (tmp)

return \_\_ffs(tmp) + offset;

offset += BITS\_PER\_LONG;

}

}

return size;

}

#if 0

static void dump\_node(void \*slot, int height, int offset)

{

struct radix\_tree\_node \*node;

int i;

if (!slot)

return;

if (height == 0) {

pr\_debug("radix entry %p offset %d\n", slot, offset);

return;

}

node = indirect\_to\_ptr(slot);

pr\_debug("radix node: %p offset %d tags %lx %lx %lx path %x count %d parent %p\n",

slot, offset, node->tags[0][0], node->tags[1][0],

node->tags[2][0], node->path, node->count, node->parent);

for (i = 0; i < RADIX\_TREE\_MAP\_SIZE; i++)

dump\_node(node->slots[i], height - 1, i);

}

/\* For debug \*/

static void radix\_tree\_dump(struct radix\_tree\_root \*root)

{

pr\_debug("radix root: %p height %d rnode %p tags %x\n",

root, root->height, root->rnode,

root->gfp\_mask >> \_\_GFP\_BITS\_SHIFT);

if (!radix\_tree\_is\_indirect\_ptr(root->rnode))

return;

dump\_node(root->rnode, root->height, 0);

}

#endif

/\*

\* This assumes that the caller has performed appropriate preallocation, and

\* that the caller has pinned this thread of control to the current CPU.

\*/

static struct radix\_tree\_node \*

radix\_tree\_node\_alloc(struct radix\_tree\_root \*root)

{

struct radix\_tree\_node \*ret = NULL;

gfp\_t gfp\_mask = root\_gfp\_mask(root);

/\*

\* Preload code isn't irq safe and it doesn't make sence to use

\* preloading in the interrupt anyway as all the allocations have to

\* be atomic. So just do normal allocation when in interrupt.

\*/

if (!gfpflags\_allow\_blocking(gfp\_mask) && !in\_interrupt()) {

struct radix\_tree\_preload \*rtp;

/\*

\* Even if the caller has preloaded, try to allocate from the

\* cache first for the new node to get accounted.

\*/

ret = kmem\_cache\_alloc(radix\_tree\_node\_cachep,

gfp\_mask | \_\_GFP\_ACCOUNT | \_\_GFP\_NOWARN);

if (ret)

goto out;

/\*

\* Provided the caller has preloaded here, we will always

\* succeed in getting a node here (and never reach

\* kmem\_cache\_alloc)

\*/

rtp = this\_cpu\_ptr(&radix\_tree\_preloads);

if (rtp->nr) {

ret = rtp->nodes;

rtp->nodes = ret->private\_data;

ret->private\_data = NULL;

rtp->nr--;

}

/\*

\* Update the allocation stack trace as this is more useful

\* for debugging.

\*/

kmemleak\_update\_trace(ret);

goto out;

}

ret = kmem\_cache\_alloc(radix\_tree\_node\_cachep,

gfp\_mask | \_\_GFP\_ACCOUNT);

out:

BUG\_ON(radix\_tree\_is\_indirect\_ptr(ret));

return ret;

}

static void radix\_tree\_node\_rcu\_free(struct rcu\_head \*head)

{

struct radix\_tree\_node \*node =

container\_of(head, struct radix\_tree\_node, rcu\_head);

int i;

/\*

\* must only free zeroed nodes into the slab. radix\_tree\_shrink

\* can leave us with a non-NULL entry in the first slot, so clear

\* that here to make sure.

\*/

for (i = 0; i < RADIX\_TREE\_MAX\_TAGS; i++)

tag\_clear(node, i, 0);

node->slots[0] = NULL;

node->count = 0;

kmem\_cache\_free(radix\_tree\_node\_cachep, node);

}

static inline void

radix\_tree\_node\_free(struct radix\_tree\_node \*node)

{

call\_rcu(&node->rcu\_head, radix\_tree\_node\_rcu\_free);

}

/\*

\* Load up this CPU's radix\_tree\_node buffer with sufficient objects to

\* ensure that the addition of a single element in the tree cannot fail. On

\* success, return zero, with preemption disabled. On error, return -ENOMEM

\* with preemption not disabled.

\*

\* To make use of this facility, the radix tree must be initialised without

\* \_\_GFP\_DIRECT\_RECLAIM being passed to INIT\_RADIX\_TREE().

\*/

static int \_\_radix\_tree\_preload(gfp\_t gfp\_mask)

{

struct radix\_tree\_preload \*rtp;

struct radix\_tree\_node \*node;

int ret = -ENOMEM;

preempt\_disable();

rtp = this\_cpu\_ptr(&radix\_tree\_preloads);

while (rtp->nr < RADIX\_TREE\_PRELOAD\_SIZE) {

preempt\_enable();

node = kmem\_cache\_alloc(radix\_tree\_node\_cachep, gfp\_mask);

if (node == NULL)

goto out;

preempt\_disable();

rtp = this\_cpu\_ptr(&radix\_tree\_preloads);

if (rtp->nr < RADIX\_TREE\_PRELOAD\_SIZE) {

node->private\_data = rtp->nodes;

rtp->nodes = node;

rtp->nr++;

} else {

kmem\_cache\_free(radix\_tree\_node\_cachep, node);

}

}

ret = 0;

out:

return ret;

}

/\*

\* Load up this CPU's radix\_tree\_node buffer with sufficient objects to

\* ensure that the addition of a single element in the tree cannot fail. On

\* success, return zero, with preemption disabled. On error, return -ENOMEM

\* with preemption not disabled.

\*

\* To make use of this facility, the radix tree must be initialised without

\* \_\_GFP\_DIRECT\_RECLAIM being passed to INIT\_RADIX\_TREE().

\*/

int radix\_tree\_preload(gfp\_t gfp\_mask)

{

/\* Warn on non-sensical use... \*/

WARN\_ON\_ONCE(!gfpflags\_allow\_blocking(gfp\_mask));

return \_\_radix\_tree\_preload(gfp\_mask);

}

EXPORT\_SYMBOL(radix\_tree\_preload);

/\*

\* The same as above function, except we don't guarantee preloading happens.

\* We do it, if we decide it helps. On success, return zero with preemption

\* disabled. On error, return -ENOMEM with preemption not disabled.

\*/

int radix\_tree\_maybe\_preload(gfp\_t gfp\_mask)

{

if (gfpflags\_allow\_blocking(gfp\_mask))

return \_\_radix\_tree\_preload(gfp\_mask);

/\* Preloading doesn't help anything with this gfp mask, skip it \*/

preempt\_disable();

return 0;

}

EXPORT\_SYMBOL(radix\_tree\_maybe\_preload);

/\*

\* Return the maximum key which can be store into a

\* radix tree with height HEIGHT.

\*/

static inline unsigned long radix\_tree\_maxindex(unsigned int height)

{

return height\_to\_maxindex[height];

}

/\*

\* Extend a radix tree so it can store key @index.

\*/

static int radix\_tree\_extend(struct radix\_tree\_root \*root,

unsigned long index, unsigned order)

{

struct radix\_tree\_node \*node;

struct radix\_tree\_node \*slot;

unsigned int height;

int tag;

/\* Figure out what the height should be. \*/

height = root->height + 1;

while (index > radix\_tree\_maxindex(height))

height++;

if ((root->rnode == NULL) && (order == 0)) {

root->height = height;

goto out;

}

do {

unsigned int newheight;

if (!(node = radix\_tree\_node\_alloc(root)))

return -ENOMEM;

/\* Propagate the aggregated tag info into the new root \*/

for (tag = 0; tag < RADIX\_TREE\_MAX\_TAGS; tag++) {

if (root\_tag\_get(root, tag))

tag\_set(node, tag, 0);

}

/\* Increase the height. \*/

newheight = root->height+1;

BUG\_ON(newheight & ~RADIX\_TREE\_HEIGHT\_MASK);

node->path = newheight;

node->count = 1;

node->parent = NULL;

slot = root->rnode;

if (radix\_tree\_is\_indirect\_ptr(slot) && newheight > 1) {

slot = indirect\_to\_ptr(slot);

slot->parent = node;

slot = ptr\_to\_indirect(slot);

}

node->slots[0] = slot;

node = ptr\_to\_indirect(node);

rcu\_assign\_pointer(root->rnode, node);

root->height = newheight;

} while (height > root->height);

out:

return 0;

}

/\*\*

\* \_\_radix\_tree\_create - create a slot in a radix tree

\* @root: radix tree root

\* @index: index key

\* @order: index occupies 2^order aligned slots

\* @nodep: returns node

\* @slotp: returns slot

\*

\* Create, if necessary, and return the node and slot for an item

\* at position @index in the radix tree @root.

\*

\* Until there is more than one item in the tree, no nodes are

\* allocated and @root->rnode is used as a direct slot instead of

\* pointing to a node, in which case \*@nodep will be NULL.

\*

\* Returns -ENOMEM, or 0 for success.

\*/

int \_\_radix\_tree\_create(struct radix\_tree\_root \*root, unsigned long index,

unsigned order, struct radix\_tree\_node \*\*nodep,

void \*\*\*slotp)

{

struct radix\_tree\_node \*node = NULL, \*slot;

unsigned int height, shift, offset;

int error;

BUG\_ON((0 < order) && (order < RADIX\_TREE\_MAP\_SHIFT));

/\* Make sure the tree is high enough. \*/

if (index > radix\_tree\_maxindex(root->height)) {

error = radix\_tree\_extend(root, index, order);

if (error)

return error;

}

slot = root->rnode;

height = root->height;

shift = height \* RADIX\_TREE\_MAP\_SHIFT;

offset = 0; /\* uninitialised var warning \*/

while (shift > order) {

if (slot == NULL) {

/\* Have to add a child node. \*/

if (!(slot = radix\_tree\_node\_alloc(root)))

return -ENOMEM;

slot->path = height;

slot->parent = node;

if (node) {

rcu\_assign\_pointer(node->slots[offset],

ptr\_to\_indirect(slot));

node->count++;

slot->path |= offset << RADIX\_TREE\_HEIGHT\_SHIFT;

} else

rcu\_assign\_pointer(root->rnode,

ptr\_to\_indirect(slot));

} else if (!radix\_tree\_is\_indirect\_ptr(slot))

break;

/\* Go a level down \*/

height--;

shift -= RADIX\_TREE\_MAP\_SHIFT;

offset = (index >> shift) & RADIX\_TREE\_MAP\_MASK;

node = indirect\_to\_ptr(slot);

slot = node->slots[offset];

}

/\* Insert pointers to the canonical entry \*/

if ((shift - order) > 0) {

int i, n = 1 << (shift - order);

offset = offset & ~(n - 1);

slot = ptr\_to\_indirect(&node->slots[offset]);

for (i = 0; i < n; i++) {

if (node->slots[offset + i])

return -EEXIST;

}

for (i = 1; i < n; i++) {

rcu\_assign\_pointer(node->slots[offset + i], slot);

node->count++;

}

}

if (nodep)

\*nodep = node;

if (slotp)

\*slotp = node ? node->slots + offset : (void \*\*)&root->rnode;

return 0;

}

/\*\*

\* \_\_radix\_tree\_insert - insert into a radix tree

\* @root: radix tree root

\* @index: index key

\* @order: key covers the 2^order indices around index

\* @item: item to insert

\*

\* Insert an item into the radix tree at position @index.

\*/

int \_\_radix\_tree\_insert(struct radix\_tree\_root \*root, unsigned long index,

unsigned order, void \*item)

{

struct radix\_tree\_node \*node;

void \*\*slot;

int error;

BUG\_ON(radix\_tree\_is\_indirect\_ptr(item));

error = \_\_radix\_tree\_create(root, index, order, &node, &slot);

if (error)

return error;

if (\*slot != NULL)

return -EEXIST;

rcu\_assign\_pointer(\*slot, item);

if (node) {

node->count++;

BUG\_ON(tag\_get(node, 0, index & RADIX\_TREE\_MAP\_MASK));

BUG\_ON(tag\_get(node, 1, index & RADIX\_TREE\_MAP\_MASK));

} else {

BUG\_ON(root\_tag\_get(root, 0));

BUG\_ON(root\_tag\_get(root, 1));

}

return 0;

}

EXPORT\_SYMBOL(\_\_radix\_tree\_insert);

/\*\*

\* \_\_radix\_tree\_lookup - lookup an item in a radix tree

\* @root: radix tree root

\* @index: index key

\* @nodep: returns node

\* @slotp: returns slot

\*

\* Lookup and return the item at position @index in the radix

\* tree @root.

\*

\* Until there is more than one item in the tree, no nodes are

\* allocated and @root->rnode is used as a direct slot instead of

\* pointing to a node, in which case \*@nodep will be NULL.

\*/

void \*\_\_radix\_tree\_lookup(struct radix\_tree\_root \*root, unsigned long index,

struct radix\_tree\_node \*\*nodep, void \*\*\*slotp)

{

struct radix\_tree\_node \*node, \*parent;

unsigned int height, shift;

void \*\*slot;

node = rcu\_dereference\_raw(root->rnode);

if (node == NULL)

return NULL;

if (!radix\_tree\_is\_indirect\_ptr(node)) {

if (index > 0)

return NULL;

if (nodep)

\*nodep = NULL;

if (slotp)

\*slotp = (void \*\*)&root->rnode;

return node;

}

node = indirect\_to\_ptr(node);

height = node->path & RADIX\_TREE\_HEIGHT\_MASK;

if (index > radix\_tree\_maxindex(height))

return NULL;

shift = (height-1) \* RADIX\_TREE\_MAP\_SHIFT;

do {

parent = node;

slot = node->slots + ((index >> shift) & RADIX\_TREE\_MAP\_MASK);

node = rcu\_dereference\_raw(\*slot);

if (node == NULL)

return NULL;

if (!radix\_tree\_is\_indirect\_ptr(node))

break;

node = indirect\_to\_ptr(node);

shift -= RADIX\_TREE\_MAP\_SHIFT;

height--;

} while (height > 0);

if (nodep)

\*nodep = parent;

if (slotp)

\*slotp = slot;

return node;

}

/\*\*

\* radix\_tree\_lookup\_slot - lookup a slot in a radix tree

\* @root: radix tree root

\* @index: index key

\*

\* Returns: the slot corresponding to the position @index in the

\* radix tree @root. This is useful for update-if-exists operations.

\*

\* This function can be called under rcu\_read\_lock iff the slot is not

\* modified by radix\_tree\_replace\_slot, otherwise it must be called

\* exclusive from other writers. Any dereference of the slot must be done

\* using radix\_tree\_deref\_slot.

\*/

void \*\*radix\_tree\_lookup\_slot(struct radix\_tree\_root \*root, unsigned long index)

{

void \*\*slot;

if (!\_\_radix\_tree\_lookup(root, index, NULL, &slot))

return NULL;

return slot;

}

EXPORT\_SYMBOL(radix\_tree\_lookup\_slot);

/\*\*

\* radix\_tree\_lookup - perform lookup operation on a radix tree

\* @root: radix tree root

\* @index: index key

\*

\* Lookup the item at the position @index in the radix tree @root.

\*

\* This function can be called under rcu\_read\_lock, however the caller

\* must manage lifetimes of leaf nodes (eg. RCU may also be used to free

\* them safely). No RCU barriers are required to access or modify the

\* returned item, however.

\*/

void \*radix\_tree\_lookup(struct radix\_tree\_root \*root, unsigned long index)

{

return \_\_radix\_tree\_lookup(root, index, NULL, NULL);

}

EXPORT\_SYMBOL(radix\_tree\_lookup);

/\*\*

\* radix\_tree\_tag\_set - set a tag on a radix tree node

\* @root: radix tree root

\* @index: index key

\* @tag: tag index

\*

\* Set the search tag (which must be < RADIX\_TREE\_MAX\_TAGS)

\* corresponding to @index in the radix tree. From

\* the root all the way down to the leaf node.

\*

\* Returns the address of the tagged item. Setting a tag on a not-present

\* item is a bug.

\*/

void \*radix\_tree\_tag\_set(struct radix\_tree\_root \*root,

unsigned long index, unsigned int tag)

{

unsigned int height, shift;

struct radix\_tree\_node \*slot;

height = root->height;

BUG\_ON(index > radix\_tree\_maxindex(height));

slot = indirect\_to\_ptr(root->rnode);

shift = (height - 1) \* RADIX\_TREE\_MAP\_SHIFT;

while (height > 0) {

int offset;

offset = (index >> shift) & RADIX\_TREE\_MAP\_MASK;

if (!tag\_get(slot, tag, offset))

tag\_set(slot, tag, offset);

slot = slot->slots[offset];

BUG\_ON(slot == NULL);

if (!radix\_tree\_is\_indirect\_ptr(slot))

break;

slot = indirect\_to\_ptr(slot);

shift -= RADIX\_TREE\_MAP\_SHIFT;

height--;

}

/\* set the root's tag bit \*/

if (slot && !root\_tag\_get(root, tag))

root\_tag\_set(root, tag);

return slot;

}

EXPORT\_SYMBOL(radix\_tree\_tag\_set);

/\*\*

\* radix\_tree\_tag\_clear - clear a tag on a radix tree node

\* @root: radix tree root

\* @index: index key

\* @tag: tag index

\*

\* Clear the search tag (which must be < RADIX\_TREE\_MAX\_TAGS)

\* corresponding to @index in the radix tree. If

\* this causes the leaf node to have no tags set then clear the tag in the

\* next-to-leaf node, etc.

\*

\* Returns the address of the tagged item on success, else NULL. ie:

\* has the same return value and semantics as radix\_tree\_lookup().

\*/

void \*radix\_tree\_tag\_clear(struct radix\_tree\_root \*root,

unsigned long index, unsigned int tag)

{

struct radix\_tree\_node \*node = NULL;

struct radix\_tree\_node \*slot = NULL;

unsigned int height, shift;

int uninitialized\_var(offset);

height = root->height;

if (index > radix\_tree\_maxindex(height))

goto out;

shift = height \* RADIX\_TREE\_MAP\_SHIFT;

slot = root->rnode;

while (shift) {

if (slot == NULL)

goto out;

if (!radix\_tree\_is\_indirect\_ptr(slot))

break;

slot = indirect\_to\_ptr(slot);

shift -= RADIX\_TREE\_MAP\_SHIFT;

offset = (index >> shift) & RADIX\_TREE\_MAP\_MASK;

node = slot;

slot = slot->slots[offset];

}

if (slot == NULL)

goto out;

while (node) {

if (!tag\_get(node, tag, offset))

goto out;

tag\_clear(node, tag, offset);

if (any\_tag\_set(node, tag))

goto out;

index >>= RADIX\_TREE\_MAP\_SHIFT;

offset = index & RADIX\_TREE\_MAP\_MASK;

node = node->parent;

}

/\* clear the root's tag bit \*/

if (root\_tag\_get(root, tag))

root\_tag\_clear(root, tag);

out:

return slot;

}

EXPORT\_SYMBOL(radix\_tree\_tag\_clear);

/\*\*

\* radix\_tree\_tag\_get - get a tag on a radix tree node

\* @root: radix tree root

\* @index: index key

\* @tag: tag index (< RADIX\_TREE\_MAX\_TAGS)

\*

\* Return values:

\*

\* 0: tag not present or not set

\* 1: tag set

\*

\* Note that the return value of this function may not be relied on, even if

\* the RCU lock is held, unless tag modification and node deletion are excluded

\* from concurrency.

\*/

int radix\_tree\_tag\_get(struct radix\_tree\_root \*root,

unsigned long index, unsigned int tag)

{

unsigned int height, shift;

struct radix\_tree\_node \*node;

/\* check the root's tag bit \*/

if (!root\_tag\_get(root, tag))

return 0;

node = rcu\_dereference\_raw(root->rnode);

if (node == NULL)

return 0;

if (!radix\_tree\_is\_indirect\_ptr(node))

return (index == 0);

node = indirect\_to\_ptr(node);

height = node->path & RADIX\_TREE\_HEIGHT\_MASK;

if (index > radix\_tree\_maxindex(height))

return 0;

shift = (height - 1) \* RADIX\_TREE\_MAP\_SHIFT;

for ( ; ; ) {

int offset;

if (node == NULL)

return 0;

node = indirect\_to\_ptr(node);

offset = (index >> shift) & RADIX\_TREE\_MAP\_MASK;

if (!tag\_get(node, tag, offset))

return 0;

if (height == 1)

return 1;

node = rcu\_dereference\_raw(node->slots[offset]);

if (!radix\_tree\_is\_indirect\_ptr(node))

return 1;

shift -= RADIX\_TREE\_MAP\_SHIFT;

height--;

}

}

EXPORT\_SYMBOL(radix\_tree\_tag\_get);

/\*\*

\* radix\_tree\_next\_chunk - find next chunk of slots for iteration

\*

\* @root: radix tree root

\* @iter: iterator state

\* @flags: RADIX\_TREE\_ITER\_\* flags and tag index

\* Returns: pointer to chunk first slot, or NULL if iteration is over

\*/

void \*\*radix\_tree\_next\_chunk(struct radix\_tree\_root \*root,

struct radix\_tree\_iter \*iter, unsigned flags)

{

unsigned shift, tag = flags & RADIX\_TREE\_ITER\_TAG\_MASK;

struct radix\_tree\_node \*rnode, \*node;

unsigned long index, offset, height;

if ((flags & RADIX\_TREE\_ITER\_TAGGED) && !root\_tag\_get(root, tag))

return NULL;

/\*

\* Catch next\_index overflow after ~0UL. iter->index never overflows

\* during iterating; it can be zero only at the beginning.

\* And we cannot overflow iter->next\_index in a single step,

\* because RADIX\_TREE\_MAP\_SHIFT < BITS\_PER\_LONG.

\*

\* This condition also used by radix\_tree\_next\_slot() to stop

\* contiguous iterating, and forbid swithing to the next chunk.

\*/

index = iter->next\_index;

if (!index && iter->index)

return NULL;

rnode = rcu\_dereference\_raw(root->rnode);

if (radix\_tree\_is\_indirect\_ptr(rnode)) {

rnode = indirect\_to\_ptr(rnode);

} else if (rnode && !index) {

/\* Single-slot tree \*/

iter->index = 0;

iter->next\_index = 1;

iter->tags = 1;

return (void \*\*)&root->rnode;

} else

return NULL;

restart:

height = rnode->path & RADIX\_TREE\_HEIGHT\_MASK;

shift = (height - 1) \* RADIX\_TREE\_MAP\_SHIFT;

offset = index >> shift;

/\* Index outside of the tree \*/

if (offset >= RADIX\_TREE\_MAP\_SIZE)

return NULL;

node = rnode;

while (1) {

struct radix\_tree\_node \*slot;

if ((flags & RADIX\_TREE\_ITER\_TAGGED) ?

!test\_bit(offset, node->tags[tag]) :

!node->slots[offset]) {

/\* Hole detected \*/

if (flags & RADIX\_TREE\_ITER\_CONTIG)

return NULL;

if (flags & RADIX\_TREE\_ITER\_TAGGED)

offset = radix\_tree\_find\_next\_bit(

node->tags[tag],

RADIX\_TREE\_MAP\_SIZE,

offset + 1);

else

while (++offset < RADIX\_TREE\_MAP\_SIZE) {

if (node->slots[offset])

break;

}

index &= ~((RADIX\_TREE\_MAP\_SIZE << shift) - 1);

index += offset << shift;

/\* Overflow after ~0UL \*/

if (!index)

return NULL;

if (offset == RADIX\_TREE\_MAP\_SIZE)

goto restart;

}

/\* This is leaf-node \*/

if (!shift)

break;

slot = rcu\_dereference\_raw(node->slots[offset]);

if (slot == NULL)

goto restart;

if (!radix\_tree\_is\_indirect\_ptr(slot))

break;

node = indirect\_to\_ptr(slot);

shift -= RADIX\_TREE\_MAP\_SHIFT;

offset = (index >> shift) & RADIX\_TREE\_MAP\_MASK;

}

/\* Update the iterator state \*/

iter->index = index;

iter->next\_index = (index | RADIX\_TREE\_MAP\_MASK) + 1;

/\* Construct iter->tags bit-mask from node->tags[tag] array \*/

if (flags & RADIX\_TREE\_ITER\_TAGGED) {

unsigned tag\_long, tag\_bit;

tag\_long = offset / BITS\_PER\_LONG;

tag\_bit = offset % BITS\_PER\_LONG;

iter->tags = node->tags[tag][tag\_long] >> tag\_bit;

/\* This never happens if RADIX\_TREE\_TAG\_LONGS == 1 \*/

if (tag\_long < RADIX\_TREE\_TAG\_LONGS - 1) {

/\* Pick tags from next element \*/

if (tag\_bit)

iter->tags |= node->tags[tag][tag\_long + 1] <<

(BITS\_PER\_LONG - tag\_bit);

/\* Clip chunk size, here only BITS\_PER\_LONG tags \*/

iter->next\_index = index + BITS\_PER\_LONG;

}

}

return node->slots + offset;

}

EXPORT\_SYMBOL(radix\_tree\_next\_chunk);

/\*\*

\* radix\_tree\_range\_tag\_if\_tagged - for each item in given range set given

\* tag if item has another tag set

\* @root: radix tree root

\* @first\_indexp: pointer to a starting index of a range to scan

\* @last\_index: last index of a range to scan

\* @nr\_to\_tag: maximum number items to tag

\* @iftag: tag index to test

\* @settag: tag index to set if tested tag is set

\*

\* This function scans range of radix tree from first\_index to last\_index

\* (inclusive). For each item in the range if iftag is set, the function sets

\* also settag. The function stops either after tagging nr\_to\_tag items or

\* after reaching last\_index.

\*

\* The tags must be set from the leaf level only and propagated back up the

\* path to the root. We must do this so that we resolve the full path before

\* setting any tags on intermediate nodes. If we set tags as we descend, then

\* we can get to the leaf node and find that the index that has the iftag

\* set is outside the range we are scanning. This reults in dangling tags and

\* can lead to problems with later tag operations (e.g. livelocks on lookups).

\*

\* The function returns number of leaves where the tag was set and sets

\* \*first\_indexp to the first unscanned index.

\* WARNING! \*first\_indexp can wrap if last\_index is ULONG\_MAX. Caller must

\* be prepared to handle that.

\*/

unsigned long radix\_tree\_range\_tag\_if\_tagged(struct radix\_tree\_root \*root,

unsigned long \*first\_indexp, unsigned long last\_index,

unsigned long nr\_to\_tag,

unsigned int iftag, unsigned int settag)

{

unsigned int height = root->height;

struct radix\_tree\_node \*node = NULL;

struct radix\_tree\_node \*slot;

unsigned int shift;

unsigned long tagged = 0;

unsigned long index = \*first\_indexp;

last\_index = min(last\_index, radix\_tree\_maxindex(height));

if (index > last\_index)

return 0;

if (!nr\_to\_tag)

return 0;

if (!root\_tag\_get(root, iftag)) {

\*first\_indexp = last\_index + 1;

return 0;

}

if (height == 0) {

\*first\_indexp = last\_index + 1;

root\_tag\_set(root, settag);

return 1;

}

shift = (height - 1) \* RADIX\_TREE\_MAP\_SHIFT;

slot = indirect\_to\_ptr(root->rnode);

for (;;) {

unsigned long upindex;

int offset;

offset = (index >> shift) & RADIX\_TREE\_MAP\_MASK;

if (!slot->slots[offset])

goto next;

if (!tag\_get(slot, iftag, offset))

goto next;

if (shift) {

node = slot;

slot = slot->slots[offset];

if (radix\_tree\_is\_indirect\_ptr(slot)) {

slot = indirect\_to\_ptr(slot);

shift -= RADIX\_TREE\_MAP\_SHIFT;

continue;

} else {

slot = node;

node = node->parent;

}

}

/\* tag the leaf \*/

tagged += 1 << shift;

tag\_set(slot, settag, offset);

/\* walk back up the path tagging interior nodes \*/

upindex = index;

while (node) {

upindex >>= RADIX\_TREE\_MAP\_SHIFT;

offset = upindex & RADIX\_TREE\_MAP\_MASK;

/\* stop if we find a node with the tag already set \*/

if (tag\_get(node, settag, offset))

break;

tag\_set(node, settag, offset);

node = node->parent;

}

/\*

\* Small optimization: now clear that node pointer.

\* Since all of this slot's ancestors now have the tag set

\* from setting it above, we have no further need to walk

\* back up the tree setting tags, until we update slot to

\* point to another radix\_tree\_node.

\*/

node = NULL;

next:

/\* Go to next item at level determined by 'shift' \*/

index = ((index >> shift) + 1) << shift;

/\* Overflow can happen when last\_index is ~0UL... \*/

if (index > last\_index || !index)

break;

if (tagged >= nr\_to\_tag)

break;

while (((index >> shift) & RADIX\_TREE\_MAP\_MASK) == 0) {

/\*

\* We've fully scanned this node. Go up. Because

\* last\_index is guaranteed to be in the tree, what

\* we do below cannot wander astray.

\*/

slot = slot->parent;

shift += RADIX\_TREE\_MAP\_SHIFT;

}

}

/\*

\* We need not to tag the root tag if there is no tag which is set with

\* settag within the range from \*first\_indexp to last\_index.

\*/

if (tagged > 0)

root\_tag\_set(root, settag);

\*first\_indexp = index;

return tagged;

}

EXPORT\_SYMBOL(radix\_tree\_range\_tag\_if\_tagged);

/\*\*

\* radix\_tree\_gang\_lookup - perform multiple lookup on a radix tree

\* @root: radix tree root

\* @results: where the results of the lookup are placed

\* @first\_index: start the lookup from this key

\* @max\_items: place up to this many items at \*results

\*

\* Performs an index-ascending scan of the tree for present items. Places

\* them at \*@results and returns the number of items which were placed at

\* \*@results.

\*

\* The implementation is naive.

\*

\* Like radix\_tree\_lookup, radix\_tree\_gang\_lookup may be called under

\* rcu\_read\_lock. In this case, rather than the returned results being

\* an atomic snapshot of the tree at a single point in time, the semantics

\* of an RCU protected gang lookup are as though multiple radix\_tree\_lookups

\* have been issued in individual locks, and results stored in 'results'.

\*/

unsigned int

radix\_tree\_gang\_lookup(struct radix\_tree\_root \*root, void \*\*results,

unsigned long first\_index, unsigned int max\_items)

{

struct radix\_tree\_iter iter;

void \*\*slot;

unsigned int ret = 0;

if (unlikely(!max\_items))

return 0;

radix\_tree\_for\_each\_slot(slot, root, &iter, first\_index) {

results[ret] = rcu\_dereference\_raw(\*slot);

if (!results[ret])

continue;

if (radix\_tree\_is\_indirect\_ptr(results[ret])) {

slot = radix\_tree\_iter\_retry(&iter);

continue;

}

if (++ret == max\_items)

break;

}

return ret;

}

EXPORT\_SYMBOL(radix\_tree\_gang\_lookup);

/\*\*

\* radix\_tree\_gang\_lookup\_slot - perform multiple slot lookup on radix tree

\* @root: radix tree root

\* @results: where the results of the lookup are placed

\* @indices: where their indices should be placed (but usually NULL)

\* @first\_index: start the lookup from this key

\* @max\_items: place up to this many items at \*results

\*

\* Performs an index-ascending scan of the tree for present items. Places

\* their slots at \*@results and returns the number of items which were

\* placed at \*@results.

\*

\* The implementation is naive.

\*

\* Like radix\_tree\_gang\_lookup as far as RCU and locking goes. Slots must

\* be dereferenced with radix\_tree\_deref\_slot, and if using only RCU

\* protection, radix\_tree\_deref\_slot may fail requiring a retry.

\*/

unsigned int

radix\_tree\_gang\_lookup\_slot(struct radix\_tree\_root \*root,

void \*\*\*results, unsigned long \*indices,

unsigned long first\_index, unsigned int max\_items)

{

struct radix\_tree\_iter iter;

void \*\*slot;

unsigned int ret = 0;

if (unlikely(!max\_items))

return 0;

radix\_tree\_for\_each\_slot(slot, root, &iter, first\_index) {

results[ret] = slot;

if (indices)

indices[ret] = iter.index;

if (++ret == max\_items)

break;

}

return ret;

}

EXPORT\_SYMBOL(radix\_tree\_gang\_lookup\_slot);

/\*\*

\* radix\_tree\_gang\_lookup\_tag - perform multiple lookup on a radix tree

\* based on a tag

\* @root: radix tree root

\* @results: where the results of the lookup are placed

\* @first\_index: start the lookup from this key

\* @max\_items: place up to this many items at \*results

\* @tag: the tag index (< RADIX\_TREE\_MAX\_TAGS)

\*

\* Performs an index-ascending scan of the tree for present items which

\* have the tag indexed by @tag set. Places the items at \*@results and

\* returns the number of items which were placed at \*@results.

\*/

unsigned int

radix\_tree\_gang\_lookup\_tag(struct radix\_tree\_root \*root, void \*\*results,

unsigned long first\_index, unsigned int max\_items,

unsigned int tag)

{

struct radix\_tree\_iter iter;

void \*\*slot;

unsigned int ret = 0;

if (unlikely(!max\_items))

return 0;

radix\_tree\_for\_each\_tagged(slot, root, &iter, first\_index, tag) {

results[ret] = rcu\_dereference\_raw(\*slot);

if (!results[ret])

continue;

if (radix\_tree\_is\_indirect\_ptr(results[ret])) {

slot = radix\_tree\_iter\_retry(&iter);

continue;

}

if (++ret == max\_items)

break;

}

return ret;

}

EXPORT\_SYMBOL(radix\_tree\_gang\_lookup\_tag);

/\*\*

\* radix\_tree\_gang\_lookup\_tag\_slot - perform multiple slot lookup on a

\* radix tree based on a tag

\* @root: radix tree root

\* @results: where the results of the lookup are placed

\* @first\_index: start the lookup from this key

\* @max\_items: place up to this many items at \*results

\* @tag: the tag index (< RADIX\_TREE\_MAX\_TAGS)

\*

\* Performs an index-ascending scan of the tree for present items which

\* have the tag indexed by @tag set. Places the slots at \*@results and

\* returns the number of slots which were placed at \*@results.

\*/

unsigned int

radix\_tree\_gang\_lookup\_tag\_slot(struct radix\_tree\_root \*root, void \*\*\*results,

unsigned long first\_index, unsigned int max\_items,

unsigned int tag)

{

struct radix\_tree\_iter iter;

void \*\*slot;

unsigned int ret = 0;

if (unlikely(!max\_items))

return 0;

radix\_tree\_for\_each\_tagged(slot, root, &iter, first\_index, tag) {

results[ret] = slot;

if (++ret == max\_items)

break;

}

return ret;

}

EXPORT\_SYMBOL(radix\_tree\_gang\_lookup\_tag\_slot);

#if defined(CONFIG\_SHMEM) && defined(CONFIG\_SWAP)

#include <linux/sched.h> /\* for cond\_resched() \*/

/\*

\* This linear search is at present only useful to shmem\_unuse\_inode().

\*/

static unsigned long \_\_locate(struct radix\_tree\_node \*slot, void \*item,

unsigned long index, unsigned long \*found\_index)

{

unsigned int shift, height;

unsigned long i;

height = slot->path & RADIX\_TREE\_HEIGHT\_MASK;

shift = (height-1) \* RADIX\_TREE\_MAP\_SHIFT;

for ( ; height > 1; height--) {

i = (index >> shift) & RADIX\_TREE\_MAP\_MASK;

for (;;) {

if (slot->slots[i] != NULL)

break;

index &= ~((1UL << shift) - 1);

index += 1UL << shift;

if (index == 0)

goto out; /\* 32-bit wraparound \*/

i++;

if (i == RADIX\_TREE\_MAP\_SIZE)

goto out;

}

slot = rcu\_dereference\_raw(slot->slots[i]);

if (slot == NULL)

goto out;

if (!radix\_tree\_is\_indirect\_ptr(slot)) {

if (slot == item) {

\*found\_index = index + i;

index = 0;

} else {

index += shift;

}

goto out;

}

slot = indirect\_to\_ptr(slot);

shift -= RADIX\_TREE\_MAP\_SHIFT;

}

/\* Bottom level: check items \*/

for (i = 0; i < RADIX\_TREE\_MAP\_SIZE; i++) {

if (slot->slots[i] == item) {

\*found\_index = index + i;

index = 0;

goto out;

}

}

index += RADIX\_TREE\_MAP\_SIZE;

out:

return index;

}

/\*\*

\* radix\_tree\_locate\_item - search through radix tree for item

\* @root: radix tree root

\* @item: item to be found

\*

\* Returns index where item was found, or -1 if not found.

\* Caller must hold no lock (since this time-consuming function needs

\* to be preemptible), and must check afterwards if item is still there.

\*/

unsigned long radix\_tree\_locate\_item(struct radix\_tree\_root \*root, void \*item)

{

struct radix\_tree\_node \*node;

unsigned long max\_index;

unsigned long cur\_index = 0;

unsigned long found\_index = -1;

do {

rcu\_read\_lock();

node = rcu\_dereference\_raw(root->rnode);

if (!radix\_tree\_is\_indirect\_ptr(node)) {

rcu\_read\_unlock();

if (node == item)

found\_index = 0;

break;

}

node = indirect\_to\_ptr(node);

max\_index = radix\_tree\_maxindex(node->path &

RADIX\_TREE\_HEIGHT\_MASK);

if (cur\_index > max\_index) {

rcu\_read\_unlock();

break;

}

cur\_index = \_\_locate(node, item, cur\_index, &found\_index);

rcu\_read\_unlock();

cond\_resched();

} while (cur\_index != 0 && cur\_index <= max\_index);

return found\_index;

}

#else

unsigned long radix\_tree\_locate\_item(struct radix\_tree\_root \*root, void \*item)

{

return -1;

}

#endif /\* CONFIG\_SHMEM && CONFIG\_SWAP \*/

/\*\*

\* radix\_tree\_shrink - shrink height of a radix tree to minimal

\* @root radix tree root

\*/

static inline void radix\_tree\_shrink(struct radix\_tree\_root \*root)

{

/\* try to shrink tree height \*/

while (root->height > 0) {

struct radix\_tree\_node \*to\_free = root->rnode;

struct radix\_tree\_node \*slot;

BUG\_ON(!radix\_tree\_is\_indirect\_ptr(to\_free));

to\_free = indirect\_to\_ptr(to\_free);

/\*

\* The candidate node has more than one child, or its child

\* is not at the leftmost slot, or it is a multiorder entry,

\* we cannot shrink.

\*/

if (to\_free->count != 1)

break;

slot = to\_free->slots[0];

if (!slot)

break;

/\*

\* We don't need rcu\_assign\_pointer(), since we are simply

\* moving the node from one part of the tree to another: if it

\* was safe to dereference the old pointer to it

\* (to\_free->slots[0]), it will be safe to dereference the new

\* one (root->rnode) as far as dependent read barriers go.

\*/

if (root->height > 1) {

if (!radix\_tree\_is\_indirect\_ptr(slot))

break;

slot = indirect\_to\_ptr(slot);

slot->parent = NULL;

slot = ptr\_to\_indirect(slot);

}

root->rnode = slot;

root->height--;

/\*

\* We have a dilemma here. The node's slot[0] must not be

\* NULLed in case there are concurrent lookups expecting to

\* find the item. However if this was a bottom-level node,

\* then it may be subject to the slot pointer being visible

\* to callers dereferencing it. If item corresponding to

\* slot[0] is subsequently deleted, these callers would expect

\* their slot to become empty sooner or later.

\*

\* For example, lockless pagecache will look up a slot, deref

\* the page pointer, and if the page is 0 refcount it means it

\* was concurrently deleted from pagecache so try the deref

\* again. Fortunately there is already a requirement for logic

\* to retry the entire slot lookup -- the indirect pointer

\* problem (replacing direct root node with an indirect pointer

\* also results in a stale slot). So tag the slot as indirect

\* to force callers to retry.

\*/

if (root->height == 0)

\*((unsigned long \*)&to\_free->slots[0]) |=

RADIX\_TREE\_INDIRECT\_PTR;

radix\_tree\_node\_free(to\_free);

}

}

/\*\*

\* \_\_radix\_tree\_delete\_node - try to free node after clearing a slot

\* @root: radix tree root

\* @node: node containing @index

\*

\* After clearing the slot at @index in @node from radix tree

\* rooted at @root, call this function to attempt freeing the

\* node and shrinking the tree.

\*

\* Returns %true if @node was freed, %false otherwise.

\*/

bool \_\_radix\_tree\_delete\_node(struct radix\_tree\_root \*root,

struct radix\_tree\_node \*node)

{

bool deleted = false;

do {

struct radix\_tree\_node \*parent;

if (node->count) {

if (node == indirect\_to\_ptr(root->rnode)) {

radix\_tree\_shrink(root);

if (root->height == 0)

deleted = true;

}

return deleted;

}

parent = node->parent;

if (parent) {

unsigned int offset;

offset = node->path >> RADIX\_TREE\_HEIGHT\_SHIFT;

parent->slots[offset] = NULL;

parent->count--;

} else {

root\_tag\_clear\_all(root);

root->height = 0;

root->rnode = NULL;

}

radix\_tree\_node\_free(node);

deleted = true;

node = parent;

} while (node);

return deleted;

}

/\*\*

\* radix\_tree\_delete\_item - delete an item from a radix tree

\* @root: radix tree root

\* @index: index key

\* @item: expected item

\*

\* Remove @item at @index from the radix tree rooted at @root.

\*

\* Returns the address of the deleted item, or NULL if it was not present

\* or the entry at the given @index was not @item.

\*/

void \*radix\_tree\_delete\_item(struct radix\_tree\_root \*root,

unsigned long index, void \*item)

{

struct radix\_tree\_node \*node;

unsigned int offset, i;

void \*\*slot;

void \*entry;

int tag;

entry = \_\_radix\_tree\_lookup(root, index, &node, &slot);

if (!entry)

return NULL;

if (item && entry != item)

return NULL;

if (!node) {

root\_tag\_clear\_all(root);

root->rnode = NULL;

return entry;

}

offset = index & RADIX\_TREE\_MAP\_MASK;

/\*

\* Clear all tags associated with the item to be deleted.

\* This way of doing it would be inefficient, but seldom is any set.

\*/

for (tag = 0; tag < RADIX\_TREE\_MAX\_TAGS; tag++) {

if (tag\_get(node, tag, offset))

radix\_tree\_tag\_clear(root, index, tag);

}

/\* Delete any sibling slots pointing to this slot \*/

for (i = 1; offset + i < RADIX\_TREE\_MAP\_SIZE; i++) {

if (node->slots[offset + i] != ptr\_to\_indirect(slot))

break;

node->slots[offset + i] = NULL;

node->count--;

}

node->slots[offset] = NULL;

node->count--;

\_\_radix\_tree\_delete\_node(root, node);

return entry;

}

EXPORT\_SYMBOL(radix\_tree\_delete\_item);

/\*\*

\* radix\_tree\_delete - delete an item from a radix tree

\* @root: radix tree root

\* @index: index key

\*

\* Remove the item at @index from the radix tree rooted at @root.

\*

\* Returns the address of the deleted item, or NULL if it was not present.

\*/

void \*radix\_tree\_delete(struct radix\_tree\_root \*root, unsigned long index)

{

return radix\_tree\_delete\_item(root, index, NULL);

}

EXPORT\_SYMBOL(radix\_tree\_delete);

/\*\*

\* radix\_tree\_tagged - test whether any items in the tree are tagged

\* @root: radix tree root

\* @tag: tag to test

\*/

int radix\_tree\_tagged(struct radix\_tree\_root \*root, unsigned int tag)

{

return root\_tag\_get(root, tag);

}

EXPORT\_SYMBOL(radix\_tree\_tagged);

static void

radix\_tree\_node\_ctor(void \*arg)

{

struct radix\_tree\_node \*node = arg;

memset(node, 0, sizeof(\*node));

INIT\_LIST\_HEAD(&node->private\_list);

}

static \_\_init unsigned long \_\_maxindex(unsigned int height)

{

unsigned int width = height \* RADIX\_TREE\_MAP\_SHIFT;

int shift = RADIX\_TREE\_INDEX\_BITS - width;

if (shift < 0)

return ~0UL;

if (shift >= BITS\_PER\_LONG)

return 0UL;

return ~0UL >> shift;

}

static \_\_init void radix\_tree\_init\_maxindex(void)

{

unsigned int i;

for (i = 0; i < ARRAY\_SIZE(height\_to\_maxindex); i++)

height\_to\_maxindex[i] = \_\_maxindex(i);

}

static int radix\_tree\_callback(struct notifier\_block \*nfb,

unsigned long action,

void \*hcpu)

{

int cpu = (long)hcpu;

struct radix\_tree\_preload \*rtp;

struct radix\_tree\_node \*node;

/\* Free per-cpu pool of perloaded nodes \*/

if (action == CPU\_DEAD || action == CPU\_DEAD\_FROZEN) {

rtp = &per\_cpu(radix\_tree\_preloads, cpu);

while (rtp->nr) {

node = rtp->nodes;

rtp->nodes = node->private\_data;

kmem\_cache\_free(radix\_tree\_node\_cachep, node);

rtp->nr--;

}

}

return NOTIFY\_OK;

}

void \_\_init radix\_tree\_init(void)

{

radix\_tree\_node\_cachep = kmem\_cache\_create("radix\_tree\_node",

sizeof(struct radix\_tree\_node), 0,

SLAB\_PANIC | SLAB\_RECLAIM\_ACCOUNT,

radix\_tree\_node\_ctor);

radix\_tree\_init\_maxindex();

hotcpu\_notifier(radix\_tree\_callback, 0);

}

/\*

\* A fast, small, non-recursive O(nlog n) sort for the Linux kernel

\*

\* Jan 23 2005 Matt Mackall <mpm@selenic.com>

\*/

#include <linux/types.h>

#include <linux/export.h>

#include <linux/sort.h>

static int alignment\_ok(const void \*base, int align)

{

return IS\_ENABLED(CONFIG\_HAVE\_EFFICIENT\_UNALIGNED\_ACCESS) ||

((unsigned long)base & (align - 1)) == 0;

}

static void u32\_swap(void \*a, void \*b, int size)

{

u32 t = \*(u32 \*)a;

\*(u32 \*)a = \*(u32 \*)b;

\*(u32 \*)b = t;

}

static void u64\_swap(void \*a, void \*b, int size)

{

u64 t = \*(u64 \*)a;

\*(u64 \*)a = \*(u64 \*)b;

\*(u64 \*)b = t;

}

static void generic\_swap(void \*a, void \*b, int size)

{

char t;

do {

t = \*(char \*)a;

\*(char \*)a++ = \*(char \*)b;

\*(char \*)b++ = t;

} while (--size > 0);

}

/\*\*

\* sort - sort an array of elements

\* @base: pointer to data to sort

\* @num: number of elements

\* @size: size of each element

\* @cmp\_func: pointer to comparison function

\* @swap\_func: pointer to swap function or NULL

\*

\* This function does a heapsort on the given array. You may provide a

\* swap\_func function optimized to your element type.

\*

\* Sorting time is O(n log n) both on average and worst-case. While

\* qsort is about 20% faster on average, it suffers from exploitable

\* O(n\*n) worst-case behavior and extra memory requirements that make

\* it less suitable for kernel use.

\*/

void sort(void \*base, size\_t num, size\_t size,

int (\*cmp\_func)(const void \*, const void \*),

void (\*swap\_func)(void \*, void \*, int size))

{

/\* pre-scale counters for performance \*/

int i = (num/2 - 1) \* size, n = num \* size, c, r;

if (!swap\_func) {

if (size == 4 && alignment\_ok(base, 4))

swap\_func = u32\_swap;

else if (size == 8 && alignment\_ok(base, 8))

swap\_func = u64\_swap;

else

swap\_func = generic\_swap;

}

/\* heapify \*/

for ( ; i >= 0; i -= size) {

for (r = i; r \* 2 + size < n; r = c) {

c = r \* 2 + size;

if (c < n - size &&

cmp\_func(base + c, base + c + size) < 0)

c += size;

if (cmp\_func(base + r, base + c) >= 0)

break;

swap\_func(base + r, base + c, size);

}

}

/\* sort \*/

for (i = n - size; i > 0; i -= size) {

swap\_func(base, base + i, size);

for (r = 0; r \* 2 + size < i; r = c) {

c = r \* 2 + size;

if (c < i - size &&

cmp\_func(base + c, base + c + size) < 0)

c += size;

if (cmp\_func(base + r, base + c) >= 0)

break;

swap\_func(base + r, base + c, size);

}

}

}

EXPORT\_SYMBOL(sort);

#if 0

#include <linux/slab.h>

/\* a simple boot-time regression test \*/

int cmpint(const void \*a, const void \*b)

{

return \*(int \*)a - \*(int \*)b;

}

static int sort\_test(void)

{

int \*a, i, r = 1;

a = kmalloc(1000 \* sizeof(int), GFP\_KERNEL);

BUG\_ON(!a);

printk("testing sort()\n");

for (i = 0; i < 1000; i++) {

r = (r \* 725861) % 6599;

a[i] = r;

}

sort(a, 1000, sizeof(int), cmpint, NULL);

for (i = 0; i < 999; i++)

if (a[i] > a[i+1]) {

printk("sort() failed!\n");

break;

}

kfree(a);

return 0;

}

module\_init(sort\_test);

#endif

/\*

\* Unified UUID/GUID definition

\*

\* Copyright (C) 2009, Intel Corp.

\* Huang Ying <ying.huang@intel.com>

\*

\* This program is free software; you can redistribute it and/or

\* modify it under the terms of the GNU General Public License version

\* 2 as published by the Free Software Foundation;

\*

\* 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 have received a copy of the GNU General Public License

\* along with this program; if not, write to the Free Software

\* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

\*/

#include <linux/kernel.h>

#include <linux/export.h>

#include <linux/uuid.h>

#include <linux/random.h>

static void \_\_uuid\_gen\_common(\_\_u8 b[16])

{

prandom\_bytes(b, 16);

/\* reversion 0b10 \*/

b[8] = (b[8] & 0x3F) | 0x80;

}

void uuid\_le\_gen(uuid\_le \*lu)

{

\_\_uuid\_gen\_common(lu->b);

/\* version 4 : random generation \*/

lu->b[7] = (lu->b[7] & 0x0F) | 0x40;

}

EXPORT\_SYMBOL\_GPL(uuid\_le\_gen);

void uuid\_be\_gen(uuid\_be \*bu)

{

\_\_uuid\_gen\_common(bu->b);

/\* version 4 : random generation \*/

bu->b[6] = (bu->b[6] & 0x0F) | 0x40;

}

EXPORT\_SYMBOL\_GPL(uuid\_be\_gen);

https://github.com/torvalds/linux

linux/lib/list\_sort.c

#define pr\_fmt(fmt) "list\_sort\_test: " fmt

#include <linux/kernel.h>

#include <linux/bug.h>

#include <linux/compiler.h>

#include <linux/export.h>

#include <linux/string.h>

#include <linux/list\_sort.h>

#include <linux/list.h>

#define MAX\_LIST\_LENGTH\_BITS 20

/\*

\* Returns a list organized in an intermediate format suited

\* to chaining of merge() calls: null-terminated, no reserved or

\* sentinel head node, "prev" links not maintained.

\*/

static struct list\_head \*merge(void \*priv,

int (\*cmp)(void \*priv, struct list\_head \*a,

struct list\_head \*b),

struct list\_head \*a, struct list\_head \*b)

{

struct list\_head head, \*tail = &head;

while (a && b) {

/\* if equal, take 'a' -- important for sort stability \*/

if ((\*cmp)(priv, a, b) <= 0) {

tail->next = a;

a = a->next;

} else {

tail->next = b;

b = b->next;

}

tail = tail->next;

}

tail->next = a?:b;

return head.next;

}

/\*

\* Combine final list merge with restoration of standard doubly-linked

\* list structure. This approach duplicates code from merge(), but

\* runs faster than the tidier alternatives of either a separate final

\* prev-link restoration pass, or maintaining the prev links

\* throughout.

\*/

static void merge\_and\_restore\_back\_links(void \*priv,

int (\*cmp)(void \*priv, struct list\_head \*a,

struct list\_head \*b),

struct list\_head \*head,

struct list\_head \*a, struct list\_head \*b)

{

struct list\_head \*tail = head;

u8 count = 0;

while (a && b) {

/\* if equal, take 'a' -- important for sort stability \*/

if ((\*cmp)(priv, a, b) <= 0) {

tail->next = a;

a->prev = tail;

a = a->next;

} else {

tail->next = b;

b->prev = tail;

b = b->next;

}

tail = tail->next;

}

tail->next = a ? : b;

do {

/\*

\* In worst cases this loop may run many iterations.

\* Continue callbacks to the client even though no

\* element comparison is needed, so the client's cmp()

\* routine can invoke cond\_resched() periodically.

\*/

if (unlikely(!(++count)))

(\*cmp)(priv, tail->next, tail->next);

tail->next->prev = tail;

tail = tail->next;

} while (tail->next);

tail->next = head;

head->prev = tail;

}

/\*\*

\* list\_sort - sort a list

\* @priv: private data, opaque to list\_sort(), passed to @cmp

\* @head: the list to sort

\* @cmp: the elements comparison function

\*

\* This function implements "merge sort", which has O(nlog(n))

\* complexity.

\*

\* The comparison function @cmp must return a negative value if @a

\* should sort before @b, and a positive value if @a should sort after

\* @b. If @a and @b are equivalent, and their original relative

\* ordering is to be preserved, @cmp must return 0.

\*/

void list\_sort(void \*priv, struct list\_head \*head,

int (\*cmp)(void \*priv, struct list\_head \*a,

struct list\_head \*b))

{

struct list\_head \*part[MAX\_LIST\_LENGTH\_BITS+1]; /\* sorted partial lists

-- last slot is a sentinel \*/

int lev; /\* index into part[] \*/

int max\_lev = 0;

struct list\_head \*list;

if (list\_empty(head))

return;

memset(part, 0, sizeof(part));

head->prev->next = NULL;

list = head->next;

while (list) {

struct list\_head \*cur = list;

list = list->next;

cur->next = NULL;

for (lev = 0; part[lev]; lev++) {

cur = merge(priv, cmp, part[lev], cur);

part[lev] = NULL;

}

if (lev > max\_lev) {

if (unlikely(lev >= ARRAY\_SIZE(part)-1)) {

printk\_once(KERN\_DEBUG "list too long for efficiency\n");

lev--;

}

max\_lev = lev;

}

part[lev] = cur;

}

for (lev = 0; lev < max\_lev; lev++)

if (part[lev])

list = merge(priv, cmp, part[lev], list);

merge\_and\_restore\_back\_links(priv, cmp, head, part[max\_lev], list);

}

EXPORT\_SYMBOL(list\_sort);

#ifdef CONFIG\_TEST\_LIST\_SORT

#include <linux/slab.h>

#include <linux/random.h>

/\*

\* The pattern of set bits in the list length determines which cases

\* are hit in list\_sort().

\*/

#define TEST\_LIST\_LEN (512+128+2) /\* not including head \*/

#define TEST\_POISON1 0xDEADBEEF

#define TEST\_POISON2 0xA324354C

struct debug\_el {

unsigned int poison1;

struct list\_head list;

unsigned int poison2;

int value;

unsigned serial;

};

/\* Array, containing pointers to all elements in the test list \*/

static struct debug\_el \*\*elts \_\_initdata;

static int \_\_init check(struct debug\_el \*ela, struct debug\_el \*elb)

{

if (ela->serial >= TEST\_LIST\_LEN) {

pr\_err("error: incorrect serial %d\n", ela->serial);

return -EINVAL;

}

if (elb->serial >= TEST\_LIST\_LEN) {

pr\_err("error: incorrect serial %d\n", elb->serial);

return -EINVAL;

}

if (elts[ela->serial] != ela || elts[elb->serial] != elb) {

pr\_err("error: phantom element\n");

return -EINVAL;

}

if (ela->poison1 != TEST\_POISON1 || ela->poison2 != TEST\_POISON2) {

pr\_err("error: bad poison: %#x/%#x\n",

ela->poison1, ela->poison2);

return -EINVAL;

}

if (elb->poison1 != TEST\_POISON1 || elb->poison2 != TEST\_POISON2) {

pr\_err("error: bad poison: %#x/%#x\n",

elb->poison1, elb->poison2);

return -EINVAL;

}

return 0;

}

static int \_\_init cmp(void \*priv, struct list\_head \*a, struct list\_head \*b)

{

struct debug\_el \*ela, \*elb;

ela = container\_of(a, struct debug\_el, list);

elb = container\_of(b, struct debug\_el, list);

check(ela, elb);

return ela->value - elb->value;

}

static int \_\_init list\_sort\_test(void)

{

int i, count = 1, err = -ENOMEM;

struct debug\_el \*el;

struct list\_head \*cur;

LIST\_HEAD(head);

pr\_debug("start testing list\_sort()\n");

elts = kcalloc(TEST\_LIST\_LEN, sizeof(\*elts), GFP\_KERNEL);

if (!elts) {

pr\_err("error: cannot allocate memory\n");

return err;

}

for (i = 0; i < TEST\_LIST\_LEN; i++) {

el = kmalloc(sizeof(\*el), GFP\_KERNEL);

if (!el) {

pr\_err("error: cannot allocate memory\n");

goto exit;

}

/\* force some equivalencies \*/

el->value = prandom\_u32() % (TEST\_LIST\_LEN / 3);

el->serial = i;

el->poison1 = TEST\_POISON1;

el->poison2 = TEST\_POISON2;

elts[i] = el;

list\_add\_tail(&el->list, &head);

}

list\_sort(NULL, &head, cmp);

err = -EINVAL;

for (cur = head.next; cur->next != &head; cur = cur->next) {

struct debug\_el \*el1;

int cmp\_result;

if (cur->next->prev != cur) {

pr\_err("error: list is corrupted\n");

goto exit;

}

cmp\_result = cmp(NULL, cur, cur->next);

if (cmp\_result > 0) {

pr\_err("error: list is not sorted\n");

goto exit;

}

el = container\_of(cur, struct debug\_el, list);

el1 = container\_of(cur->next, struct debug\_el, list);

if (cmp\_result == 0 && el->serial >= el1->serial) {

pr\_err("error: order of equivalent elements not "

"preserved\n");

goto exit;

}

if (check(el, el1)) {

pr\_err("error: element check failed\n");

goto exit;

}

count++;

}

if (head.prev != cur) {

pr\_err("error: list is corrupted\n");

goto exit;

}

if (count != TEST\_LIST\_LEN) {

pr\_err("error: bad list length %d", count);

goto exit;

}

err = 0;

exit:

for (i = 0; i < TEST\_LIST\_LEN; i++)

kfree(elts[i]);

kfree(elts);

return err;

}

late\_initcall(list\_sort\_test);

#endif /\* CONFIG\_TEST\_LIST\_SORT \*/

linux/include/linux/llist.h

#ifndef LLIST\_H

#define LLIST\_H

/\*

\* Lock-less NULL terminated single linked list

\*

\* If there are multiple producers and multiple consumers, llist\_add

\* can be used in producers and llist\_del\_all can be used in

\* consumers. They can work simultaneously without lock. But

\* llist\_del\_first can not be used here. Because llist\_del\_first

\* depends on list->first->next does not changed if list->first is not

\* changed during its operation, but llist\_del\_first, llist\_add,

\* llist\_add (or llist\_del\_all, llist\_add, llist\_add) sequence in

\* another consumer may violate that.

\*

\* If there are multiple producers and one consumer, llist\_add can be

\* used in producers and llist\_del\_all or llist\_del\_first can be used

\* in the consumer.

\*

\* This can be summarized as follow:

\*

\* | add | del\_first | del\_all

\* add | - | - | -

\* del\_first | | L | L

\* del\_all | | | -

\*

\* Where "-" stands for no lock is needed, while "L" stands for lock

\* is needed.

\*

\* The list entries deleted via llist\_del\_all can be traversed with

\* traversing function such as llist\_for\_each etc. But the list

\* entries can not be traversed safely before deleted from the list.

\* The order of deleted entries is from the newest to the oldest added

\* one. If you want to traverse from the oldest to the newest, you

\* must reverse the order by yourself before traversing.

\*

\* The basic atomic operation of this list is cmpxchg on long. On

\* architectures that don't have NMI-safe cmpxchg implementation, the

\* list can NOT be used in NMI handlers. So code that uses the list in

\* an NMI handler should depend on CONFIG\_ARCH\_HAVE\_NMI\_SAFE\_CMPXCHG.

\*

\* Copyright 2010,2011 Intel Corp.

\* Author: Huang Ying <ying.huang@intel.com>

\*

\* This program is free software; you can redistribute it and/or

\* modify it under the terms of the GNU General Public License version

\* 2 as published by the Free Software Foundation;

\*

\* 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 have received a copy of the GNU General Public License

\* along with this program; if not, write to the Free Software

\* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

\*/

#include <linux/atomic.h>

#include <linux/kernel.h>

struct llist\_head {

struct llist\_node \*first;

};

struct llist\_node {

struct llist\_node \*next;

};

#define LLIST\_HEAD\_INIT(name) { NULL }

#define LLIST\_HEAD(name) struct llist\_head name = LLIST\_HEAD\_INIT(name)

/\*\*

\* init\_llist\_head - initialize lock-less list head

\* @head: the head for your lock-less list

\*/

static inline void init\_llist\_head(struct llist\_head \*list)

{

list->first = NULL;

}

/\*\*

\* llist\_entry - get the struct of this entry

\* @ptr: the &struct llist\_node pointer.

\* @type: the type of the struct this is embedded in.

\* @member: the name of the llist\_node within the struct.

\*/

#define llist\_entry(ptr, type, member) \

container\_of(ptr, type, member)

/\*\*

\* llist\_for\_each - iterate over some deleted entries of a lock-less list

\* @pos: the &struct llist\_node to use as a loop cursor

\* @node: the first entry of deleted list entries

\*

\* In general, some entries of the lock-less list can be traversed

\* safely only after being deleted from list, so start with an entry

\* instead of list head.

\*

\* If being used on entries deleted from lock-less list directly, the

\* traverse order is from the newest to the oldest added entry. If

\* you want to traverse from the oldest to the newest, you must

\* reverse the order by yourself before traversing.

\*/

#define llist\_for\_each(pos, node) \

for ((pos) = (node); pos; (pos) = (pos)->next)

/\*\*

\* llist\_for\_each\_entry - iterate over some deleted entries of lock-less list of given type

\* @pos: the type \* to use as a loop cursor.

\* @node: the fist entry of deleted list entries.

\* @member: the name of the llist\_node with the struct.

\*

\* In general, some entries of the lock-less list can be traversed

\* safely only after being removed from list, so start with an entry

\* instead of list head.

\*

\* If being used on entries deleted from lock-less list directly, the

\* traverse order is from the newest to the oldest added entry. If

\* you want to traverse from the oldest to the newest, you must

\* reverse the order by yourself before traversing.

\*/

#define llist\_for\_each\_entry(pos, node, member) \

for ((pos) = llist\_entry((node), typeof(\*(pos)), member); \

&(pos)->member != NULL; \

(pos) = llist\_entry((pos)->member.next, typeof(\*(pos)), member))

/\*\*

\* llist\_for\_each\_entry\_safe - iterate over some deleted entries of lock-less list of given type

\* safe against removal of list entry

\* @pos: the type \* to use as a loop cursor.

\* @n: another type \* to use as temporary storage

\* @node: the first entry of deleted list entries.

\* @member: the name of the llist\_node with the struct.

\*

\* In general, some entries of the lock-less list can be traversed

\* safely only after being removed from list, so start with an entry

\* instead of list head.

\*

\* If being used on entries deleted from lock-less list directly, the

\* traverse order is from the newest to the oldest added entry. If

\* you want to traverse from the oldest to the newest, you must

\* reverse the order by yourself before traversing.

\*/

#define llist\_for\_each\_entry\_safe(pos, n, node, member) \

for (pos = llist\_entry((node), typeof(\*pos), member); \

&pos->member != NULL && \

(n = llist\_entry(pos->member.next, typeof(\*n), member), true); \

pos = n)

/\*\*

\* llist\_empty - tests whether a lock-less list is empty

\* @head: the list to test

\*

\* Not guaranteed to be accurate or up to date. Just a quick way to

\* test whether the list is empty without deleting something from the

\* list.

\*/

static inline bool llist\_empty(const struct llist\_head \*head)

{

return ACCESS\_ONCE(head->first) == NULL;

}

static inline struct llist\_node \*llist\_next(struct llist\_node \*node)

{

return node->next;

}

extern bool llist\_add\_batch(struct llist\_node \*new\_first,

struct llist\_node \*new\_last,

struct llist\_head \*head);

/\*\*

\* llist\_add - add a new entry

\* @new: new entry to be added

\* @head: the head for your lock-less list

\*

\* Returns true if the list was empty prior to adding this entry.

\*/

static inline bool llist\_add(struct llist\_node \*new, struct llist\_head \*head)

{

return llist\_add\_batch(new, new, head);

}

/\*\*

\* llist\_del\_all - delete all entries from lock-less list

\* @head: the head of lock-less list to delete all entries

\*

\* If list is empty, return NULL, otherwise, delete all entries and

\* return the pointer to the first entry. The order of entries

\* deleted is from the newest to the oldest added one.

\*/

static inline struct llist\_node \*llist\_del\_all(struct llist\_head \*head)

{

return xchg(&head->first, NULL);

}

extern struct llist\_node \*llist\_del\_first(struct llist\_head \*head);

struct llist\_node \*llist\_reverse\_order(struct llist\_node \*head);

#endif /\* LLIST\_H \*/

linux/include/linux/hash.h

#ifndef \_LINUX\_HASH\_H

#define \_LINUX\_HASH\_H

/\* Fast hashing routine for ints, longs and pointers.

(C) 2002 Nadia Yvette Chambers, IBM \*/

/\*

\* Knuth recommends primes in approximately golden ratio to the maximum

\* integer representable by a machine word for multiplicative hashing.

\* Chuck Lever verified the effectiveness of this technique:

\* http://www.citi.umich.edu/techreports/reports/citi-tr-00-1.pdf

\*

\* These primes are chosen to be bit-sparse, that is operations on

\* them can use shifts and additions instead of multiplications for

\* machines where multiplications are slow.

\*/

#include <asm/types.h>

#include <linux/compiler.h>

/\* 2^31 + 2^29 - 2^25 + 2^22 - 2^19 - 2^16 + 1 \*/

#define GOLDEN\_RATIO\_PRIME\_32 0x9e370001UL

/\* 2^63 + 2^61 - 2^57 + 2^54 - 2^51 - 2^18 + 1 \*/

#define GOLDEN\_RATIO\_PRIME\_64 0x9e37fffffffc0001UL

#if BITS\_PER\_LONG == 32

#define GOLDEN\_RATIO\_PRIME GOLDEN\_RATIO\_PRIME\_32

#define hash\_long(val, bits) hash\_32(val, bits)

#elif BITS\_PER\_LONG == 64

#define hash\_long(val, bits) hash\_64(val, bits)

#define GOLDEN\_RATIO\_PRIME GOLDEN\_RATIO\_PRIME\_64

#else

#error Wordsize not 32 or 64

#endif

static \_\_always\_inline u64 hash\_64(u64 val, unsigned int bits)

{

u64 hash = val;

#if defined(CONFIG\_ARCH\_HAS\_FAST\_MULTIPLIER) && BITS\_PER\_LONG == 64

hash = hash \* GOLDEN\_RATIO\_PRIME\_64;

#else

/\* Sigh, gcc can't optimise this alone like it does for 32 bits. \*/

u64 n = hash;

n <<= 18;

hash -= n;

n <<= 33;

hash -= n;

n <<= 3;

hash += n;

n <<= 3;

hash -= n;

n <<= 4;

hash += n;

n <<= 2;

hash += n;

#endif

/\* High bits are more random, so use them. \*/

return hash >> (64 - bits);

}

static inline u32 hash\_32(u32 val, unsigned int bits)

{

/\* On some cpus multiply is faster, on others gcc will do shifts \*/

u32 hash = val \* GOLDEN\_RATIO\_PRIME\_32;

/\* High bits are more random, so use them. \*/

return hash >> (32 - bits);

}

static inline unsigned long hash\_ptr(const void \*ptr, unsigned int bits)

{

return hash\_long((unsigned long)ptr, bits);

}

static inline u32 hash32\_ptr(const void \*ptr)

{

unsigned long val = (unsigned long)ptr;

#if BITS\_PER\_LONG == 64

val ^= (val >> 32);

#endif

return (u32)val;

}

#endif /\* \_LINUX\_HASH\_H \*/