

[PATCH] 2/2 Prezeroing large blocks of pages during allocation Version 4

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2005-03/2475.html>

From: Mel Gorman (mel_at_csn.ul.ie)

Date: 03/07/05

To: linux-mm@kvack.org

Date: Mon, 7 Mar 2005 19:40:21 +0000 (GMT)

Changelog since V3

- o Have USERZERO and KERNZERO for different types of zero pages to avoid regressing fragmentation behavior
- o Always zero in as large as blocks as possible
- o Have a per-cpu pageset for zero-pages

Changelog since V2

- o Updated to be in sync with placement policy patches
- o Fixed critical bug where non-zeroed pages would be returned to `__GFP_ZERO`

Changelog V1

- o Initial release

This is a patch that makes a step towards merging the modified allocator for reducing fragmentation with the prezeroing of pages that is based on a discussion with Christoph. When a block has to be split to satisfy a zero-page, both buddies are zero'd, one is allocated and the other is placed on the free-list for the zero-page pool. Care is taken to make sure the pools are not accidentally fragmented by having zero pages for userspace or kernelspace allocations called USERZERO and KERNZERO.

When looking at `/proc/buddyinfo`, it will seem strange that the number of blocks free at each order for USERZERO will never get over zero. This is expected as when a page is freed, it cannot go back into the USERZERO pool on the assumption it is no longer all zeros.

I would expect that a scrubber daemon would go through the KERNORCLM and USERRCLM pools, remove pages, scrub them and move them to the appropriate zero pool. It is important that pages are moved to the correct pools to avoid fragmentation.

The patch also counts how many blocks of each order were zeroed. This gives a rough indicator if large blocks are frequently zeroed or not. I found that order-0 are the most frequent zeroed block because of the per-cpu caches. This means we rarely win with zeroing in the allocator but the

Linux-Kernel: [PATCH] 2/2 Prezeroing large blocks of pages during allocation Version 4

accounting mechanisms are still handy for the scrubber daemon.

This patch mildly regresses how well fragmentation is handled but not seriously enough to worry about at this point. The greater concern is that the raw performance of the patch is really bad, possibly due to excessive traversal of empty zero lists.

Allocation of huge pages under pressure

Order: 10

Attempted allocations: 160

Success allocs: 103

Failed allocs: 36

% Success: 64

(Using only placement policy, it is 76%)

Allocation of huge pages at rest

Order: 10

Attempted allocations: 160

Success allocs: 115

Failed allocs: 10

% Success: 71

(Using only placement policy, it is 81%)

The aim9 benchmarks are terrible;

```
mel@joshua:~$ grep _test vmregressbench-2.6.11-mbuddy-v9/aim9/log.txt
 7 page_test 60.01 4346 72.42126 123116.15 System Allocations & Pages/second
 8 brk_test 60.03 1748 29.11877 495019.16 System Memory Allocations/second
 9 jmp_test 60.00 248760 4146.00000 4146000.00 Non-local gotos/second
10 signal_test 60.01 5848 97.45042 97450.42 Signal Traps/second
11 exec_test 60.05 789 13.13905 65.70 Program Loads/second
12 fork_test 60.05 951 15.83680 1583.68 Task Creations/second
13 link_test 60.01 6277 104.59923 6589.75 Link/Unlink Pairs/second
```

```
mel@joshua:~$ grep _test vmregressbench-2.6.11-mbuddy-v9-prezero-v4/aim9/log.txt
 7 page_test 60.01 1551 25.84569 43937.68 System Allocations & Pages/second
 8 brk_test 60.01 1701 28.34528 481869.69 System Memory Allocations/second
 9 jmp_test 60.00 249948 4165.80000 4165800.00 Non-local gotos/second
10 signal_test 60.00 5732 95.53333 95533.33 Signal Traps/second
11 exec_test 60.08 733 12.20040 61.00 Program Loads/second
12 fork_test 60.04 828 13.79081 1379.08 Task Creations/second
13 link_test 60.00 6226 103.76667 6537.30 Link/Unlink Pairs/second
```

Look at the massive difference in the performance of page_test. This will need fixing before merging with anything.

This patch requires that the placement policy patch be applied first.

Signed-off-by: Mel Gorman <mel@csn.ul.ie>

```
diff -rup -X /usr/src/patchset-0.5/bin//dontdiff linux-2.6.11-mbuddy-v9/include/linux/mmzone.h
```

```
linux-2.6.11-mbuddy-v9-prezero-v4/include/linux/mmzone.h
```

```
--- linux-2.6.11-mbuddy-v9/include/linux/mmzone.h 2005-03-07 17:49:33.000000000 +0000
```

```
+++ linux-2.6.11-mbuddy-v9-prezero-v4/include/linux/mmzone.h 2005-03-07 17:50:16.000000000
```

```

+0000
@@ -19,12 +19,14 @@
#else
#define MAX_ORDER CONFIG_FORCE_MAX_ZONEORDER
#endif
-#define ALLOC_TYPES 4
-#define BITS_PER_ALLOC_TYPE 3
+#define ALLOC_TYPES 6
+#define BITS_PER_ALLOC_TYPE 5
#define ALLOC_KERNNORCLM 0
#define ALLOC_KERNRCLM 1
#define ALLOC_USERRCLM 2
#define ALLOC_FALLBACK 3
+#define ALLOC_USERZERO 4
+#define ALLOC_KERNZERO 5

struct free_area {
    struct list_head free_list;
@@ -128,6 +130,7 @@ struct zone {
    unsigned long lowmem_reserve[MAX_NR_ZONES];

    struct per_cpu_pageset pageset[NR_CPUS];
+ struct per_cpu_pageset pageset_zero[NR_CPUS];

    /*
     * free areas of different sizes
@@ -169,6 +172,7 @@ struct zone {
    unsigned long reserve_count[ALLOC_TYPES];
    unsigned long kernnorclm_full_steal;
    unsigned long kernnorclm_partial_steal;
+ unsigned long zeroblock_count[MAX_ORDER];
    unsigned long bulk_requests[MAX_ORDER];
    unsigned long bulk_allocated[MAX_ORDER];
#endif
@@ -284,6 +288,8 @@ struct zone {
#define inc_kernnorclm_full_steal(zone) zone->kernnorclm_full_steal++
#define inc_bulk_request(zone, order) zone->bulk_requests[order]++
#define inc_bulk_allocated(zone, order) zone->bulk_allocated[order]++
+#define inc_zeroblock_count(zone, order, flags) \
+ flags & __GFP_ZERO ? zone->zeroblock_count[order]++ : 0
static inline void inc_reserve_count(struct zone *zone, int type) {
    if (type == ALLOC_FALLBACK) zone->fallback_reserve++;
    zone->reserve_count[type]++;
@@ -304,6 +310,7 @@ struct zone {
static inline void dec_reserve_count(str
    zone->fallback_reserve-- : 0
#define inc_kernnorclm_partial_steal(zone) do { } while (0)
#define inc_kernnorclm_full_steal(zone) do { } while (0)
+#define inc_zeroblock_count(zone, order, flags) do { } while (0)
#define inc_bulk_request(zone, order) do { } while (0)
#define inc_bulk_allocated(zone, order) do { } while (0)
#endif

```

Linux-Kernel: [PATCH] 2/2 Prezeroing large blocks of pages during allocation Version 4

```

diff -rup -X /usr/src/patchset-0.5/bin//dontdiff linux-2.6.11-mbuddy-v9/mm/page_alloc.c
linux-2.6.11-mbuddy-v9-prezero-v4/mm/page_alloc.c
--- linux-2.6.11-mbuddy-v9/mm/page_alloc.c 2005-03-07 17:49:33.000000000 +0000
+++ linux-2.6.11-mbuddy-v9-prezero-v4/mm/page_alloc.c 2005-03-07 18:45:04.000000000 +0000
@@ -57,6 +57,15 @@ int sysctl_lowmem_reserve_ratio[MAX_NR_Z
EXPORT_SYMBOL(totalram_pages);
EXPORT_SYMBOL(nr_swap_pages);

+static inline void prep_zero_page(struct page *page, int order, int gfp_flags)
+{
+ int i;
+
+ BUG_ON((gfp_flags & (__GFP_WAIT | __GFP_HIGHMEM)) == __GFP_HIGHMEM);
+ for(i = 0; i < (1 << order); i++)
+ clear_highpage(page + i);
+}
+
+/**
+ * The allocator tries to put allocations of the same type in the
+ * same 2^MAX_ORDER blocks of pages. When memory is low, this may
@@ -70,10 +79,12 @@ EXPORT_SYMBOL(nr_swap_pages);
+
+ */
int fallback_allocs[ALLOC_TYPES][ALLOC_TYPES] = {
- {ALLOC_KERNRCLM, ALLOC_FALLBACK, ALLOC_KERNRCLM, ALLOC_USERRCLM},
- {ALLOC_KERNRCLM, ALLOC_FALLBACK, ALLOC_KERNRCLM, ALLOC_USERRCLM},
- {ALLOC_USERRCLM, ALLOC_FALLBACK, ALLOC_KERNRCLM, ALLOC_KERNRCLM},
- {ALLOC_FALLBACK, ALLOC_KERNRCLM, ALLOC_KERNRCLM, ALLOC_USERRCLM}
+ {ALLOC_KERNRCLM, ALLOC_FALLBACK, ALLOC_USERZERO, ALLOC_KERNRCLM,
+ ALLOC_USERRCLM},
+ {ALLOC_KERNRCLM, ALLOC_FALLBACK, ALLOC_KERNRCLM, ALLOC_USERZERO,
+ ALLOC_USERRCLM},
+ {ALLOC_USERRCLM, ALLOC_FALLBACK, ALLOC_KERNRCLM, ALLOC_USERZERO,
+ ALLOC_KERNRCLM},
+ {ALLOC_FALLBACK, ALLOC_KERNRCLM, ALLOC_KERNRCLM, ALLOC_USERZERO,
+ ALLOC_USERRCLM},
+ {ALLOC_USERZERO, ALLOC_USERRCLM, ALLOC_FALLBACK, ALLOC_KERNRCLM,
+ ALLOC_KERNRCLM},
+ {ALLOC_KERNZERO, ALLOC_KERNRCLM, ALLOC_FALLBACK, ALLOC_KERNRCLM,
+ ALLOC_USERRCLM}
};

/*
@@ -85,7 +96,7 @@ EXPORT_SYMBOL(zone_table);

static char *zone_names[MAX_NR_ZONES] = { "DMA", "Normal", "HighMem" };
static char *type_names[ALLOC_TYPES] = { "KernNoRclm", "KernRclm",
- "UserRclm", "Fallback" };
+ "UserRclm", "Fallback", "UserZero", "KernZero" };
int min_free_kbytes = 1024;

```

Linux-Kernel: [PATCH] 2/2 Prezeroing large blocks of pages during allocation Version 4

```
unsigned long __initdata nr_kernel_pages;
@@ -150,6 +161,12 @@ static inline int get_pageblock_type(str
    /* Bit 3 means that the block is reserved for fallbacks */
    if (test_bit(bitidx+2, zone->free_area_usemap)) return ALLOC_FALLBACK;

+ /* Bit 4 means that the block is reserved for zero-page allocations */
+ if (test_bit(bitidx+3, zone->free_area_usemap)) return ALLOC_USERZERO;
+
+ /* Bit 5 means that the block is reserved for zero-page allocations */
+ if (test_bit(bitidx+4, zone->free_area_usemap)) return ALLOC_KERNZERO;
+
    return ALLOC_KERNNORCLM;
}

@@ -167,6 +184,8 @@ static inline void set_pageblock_type(st
    set_bit(bitidx, zone->free_area_usemap);
    clear_bit(bitidx+1, zone->free_area_usemap);
    clear_bit(bitidx+2, zone->free_area_usemap);
+ clear_bit(bitidx+3, zone->free_area_usemap);
+ clear_bit(bitidx+4, zone->free_area_usemap);
    return;
}

@@ -174,6 +193,8 @@ static inline void set_pageblock_type(st
    clear_bit(bitidx, zone->free_area_usemap);
    set_bit(bitidx+1, zone->free_area_usemap);
    clear_bit(bitidx+2, zone->free_area_usemap);
+ clear_bit(bitidx+3, zone->free_area_usemap);
+ clear_bit(bitidx+4, zone->free_area_usemap);
    return;
}

@@ -181,11 +202,34 @@ static inline void set_pageblock_type(st
    clear_bit(bitidx, zone->free_area_usemap);
    clear_bit(bitidx+1, zone->free_area_usemap);
    set_bit(bitidx+2, zone->free_area_usemap);
+ clear_bit(bitidx+3, zone->free_area_usemap);
+ clear_bit(bitidx+4, zone->free_area_usemap);
+ }
+
+ if (type == ALLOC_USERZERO) {
+ clear_bit(bitidx, zone->free_area_usemap);
+ clear_bit(bitidx+1, zone->free_area_usemap);
+ clear_bit(bitidx+2, zone->free_area_usemap);
+ set_bit(bitidx+3, zone->free_area_usemap);
+ clear_bit(bitidx+4, zone->free_area_usemap);
+ return;
+ }
+
+ if (type == ALLOC_KERNZERO) {
+ clear_bit(bitidx, zone->free_area_usemap);
```

Linux-Kernel: [PATCH] 2/2 Prezeroing large blocks of pages during allocation Version 4

```

+ clear_bit(bitidx+1, zone->free_area_usemap);
+ clear_bit(bitidx+2, zone->free_area_usemap);
+ clear_bit(bitidx+3, zone->free_area_usemap);
+ set_bit(bitidx+4, zone->free_area_usemap);
+ return;
    }

+ /* KERNORCLM allocation */
    clear_bit(bitidx, zone->free_area_usemap);
    clear_bit(bitidx+1, zone->free_area_usemap);
    clear_bit(bitidx+2, zone->free_area_usemap);
+ clear_bit(bitidx+3, zone->free_area_usemap);
+ clear_bit(bitidx+4, zone->free_area_usemap);

}

@@ -344,6 +388,13 @@ static inline void __free_pages_bulk (st

    /* Select the areas to place free pages on */
    alloctype = get_pageblock_type(page);
+
+ /*
+ * Tricky, if the page was originally a zeroed page, chances are it
+ * is not any more
+ */
+ if (alloctype == ALLOC_KERNZERO) alloctype = ALLOC_KERNORCLM;
+ if (alloctype == ALLOC_USERZERO) alloctype = ALLOC_USERRCLM;
    freelist = zone->free_area_lists[alloctype];

    zone->free_pages += order_size;
@@ -539,8 +590,10 @@ static void prep_new_page(struct page *p
 * This function removes a 2**MAX_ORDER-1 block of pages from the zones
 * global list of large pages. It returns 1 on successful removal
 */
-static inline int steal_globallist(struct zone *zone, int alloctype) {
+static inline int steal_globallist(struct zone *zone, int alloctype,
+ unsigned long flags, unsigned long *irq_flags) {
    struct page *page;
+ int reserve_type = alloctype;

    /* Fail if there are no pages on the global list */
    if (list_empty(&(zone->free_area_global.free_list)))
@@ -553,9 +606,25 @@ static inline int steal_globallist(struc
    page = list_entry(zone->free_area_global.free_list.next,
        struct page, lru);
    list_del(&page->lru);
+ inc_globalsteal_count(zone);
+
+ /* Zero the whole block if this is a __GFP_ZERO allocation */
+ if (flags & __GFP_ZERO) {
+ spin_unlock_irqrestore(&zone->lock, *irq_flags);

```

Linux-Kernel: [PATCH] 2/2 Prezeroing large blocks of pages during allocation Version 4

```

+ prep_zero_page(page, MAX_ORDER-1, flags);
+ inc_zeroblock_count(zone, MAX_ORDER-1, flags);
+
+ /* Do not bother reserving zones for zeroed allocs */
+ if (flags & __GFP_USERRCLM)
+ reserve_type=ALLOC_USERRCLM;
+ else
+ reserve_type=ALLOC_KERNORCLM;
+
+ spin_lock_irqsave(&zone->lock, *irq_flags);
+ }
+
+ list_add(&page->lru,
+         &(zone->free_area_lists[alloctype][MAX_ORDER-1].free_list));
- inc_globalsteal_count(zone);

/*
 * Reserve this whole block of pages. If the number of blocks
@@ -571,18 +640,20 @@ static inline int steal_globallist(struct
+ set_pageblock_type(page, zone, ALLOC_FALLBACK);
+ inc_reserve_count(zone, ALLOC_FALLBACK);
+ } else {
- set_pageblock_type(page, zone, alloctype);
- inc_reserve_count(zone, alloctype);
+ set_pageblock_type(page, zone, reserve_type);
+ inc_reserve_count(zone, reserve_type);
+ }

return 1;

}
+
/*
 * Do the hard work of removing an element from the buddy allocator.
 * Call me with the zone->lock already held.
 */
-static struct page *__rmqueue(struct zone *zone, unsigned int order, int flags)
+static struct page *__rmqueue(struct zone *zone, unsigned int order, int flags,
+ unsigned long *irq_flags)
{
+ struct free_area * area;
+ unsigned int current_order;
@@ -594,7 +665,12 @@ static struct page *__rmqueue(struct zon
+ int alloctype, start_alloctype;
+ int retry_count=0;
+ int startorder = order;
- if (flags & __GFP_USERRCLM) {
+
+ if (flags & __GFP_ZERO && flags & __GFP_USERRCLM) {
+ alloctype = ALLOC_USERZERO;
+ } else if (flags & __GFP_ZERO) {

```

Linux-Kernel: [PATCH] 2/2 Prezeroing large blocks of pages during allocation Version 4

```

+ alloctype = ALLOC_KERNZERO;
+ } else if (flags & __GFP_USERRCLM) {
+     alloctype = ALLOC_USERRCLM;
+ }
+     else if (flags & __GFP_KERNRCLM) {
@@ -662,6 +738,27 @@ remove_page:
+         inc_reserve_count(zone, reserve_type);
+         inc_kernnorclm_full_steal(zone);
+     } else inc_kernnorclm_partial_steal(zone);
+
+ }
+
+ /* Zero out the block of pages if necessary */
+ if (flags & __GFP_ZERO &&
+ (alloctype != ALLOC_USERZERO &&
+ alloctype != ALLOC_KERNZERO)) {
+
+ /* Decide what list to put the zero pages on */
+ if (flags & __GFP_USERRCLM)
+ area = zone->free_area_lists[ALLOC_USERZERO] +
+ current_order;
+ else
+ area = zone->free_area_lists[ALLOC_KERNZERO] +
+ current_order;
+
+ /* Zero the block of pages */
+ spin_unlock_irqrestore(&zone->lock, *irq_flags);
+ prep_zero_page(page, current_order, flags);
+ inc_zeroblock_count(zone, current_order, flags);
+ spin_lock_irqsave(&zone->lock, *irq_flags);
+ }

+     return expand(zone, page, order, current_order, area);
@@ -670,7 +767,7 @@ remove_page:
+     /* Take from the global pool if this is the first attempt */
+     if (!global_split) {
+         global_split=1;
- if (steal_globallist(zone, start_alloctype)) {
+ if (steal_globallist(zone, start_alloctype, flags, irq_flags)) {
+         startorder = MAX_ORDER-1;
+         goto retry;
+     }
@@ -741,7 +838,7 @@ static int rmqueue_bulk(struct zone *zon
+     if (current_order >= MAX_ORDER) BUG();

+     /* Allocate a block at the current_order */
- page = __rmqueue(zone, current_order, gfp_flags);
+ page = __rmqueue(zone, current_order, gfp_flags, &flags);
+     if (page == NULL) {
+         if (current_order == order) break;
+         else {

```

Linux-Kernel: [PATCH] 2/2 Prezeroing large blocks of pages during allocation Version 4

```

@@ -778,9 +875,10 @@ static void __drain_pages(unsigned int c
    int i;

    for_each_zone(zone) {
- struct per_cpu_pageset *pset;
+ struct per_cpu_pageset *pset, *psetzero;

        pset = &zone->pageset[cpu];
+ psetzero = &zone->pageset_zero[cpu];
        for (i = 0; i < ARRAY_SIZE(pset->pcp); i++) {
            struct per_cpu_pages *pcp;

@@ -788,6 +886,15 @@ static void __drain_pages(unsigned int c
                pcp->count -= free_pages_bulk(zone, pcp->count,
                    &pcp->list, 0);
        }

+
+ for (i = 0; i < ARRAY_SIZE(psetzero->pcp); i++) {
+ struct per_cpu_pages *pcp;
+
+ pcp = &psetzero->pcp[i];
+ pcp->count -= free_pages_bulk(zone, pcp->count,
+ &pcp->list, 0);
+ }
+
    }
}
#endif /* CONFIG_PM || CONFIG_HOTPLUG_CPU */
@@ -898,15 +1005,6 @@ void fastcall free_cold_page(struct page
    free_hot_cold_page(page, 1);
}

-static inline void prep_zero_page(struct page *page, int order, int gfp_flags)
-{
- int i;
-
- BUG_ON((gfp_flags & (__GFP_WAIT | __GFP_HIGHMEM)) == __GFP_HIGHMEM);
- for(i = 0; i < (1 << order); i++)
- clear_highpage(page + i);
-}
-
/*
 * Really, prep_compound_page() should be called from __rmqueue_bulk(). But
 * we cheat by calling it from here, in the order > 0 path. Saves a branch
@@ -922,7 +1020,11 @@ buffered_rmqueue(struct zone *zone, int
    if (order == 0 && (gfp_flags & __GFP_USERRCLM)) {
        struct per_cpu_pages *pcp;

- pcp = &zone->pageset[get_cpu()].pcp[cold];
+ if (gfp_flags & __GFP_ZERO)
+ pcp = &zone->pageset_zero[get_cpu()].pcp[cold];

```

```

+ else
+ pcp = &zone->pageset[get_cpu()].pcp[cold];
+
    local_irq_save(flags);
    if (pcp->count <= pcp->low)
        pcp->count += rmqueue_bulk(zone, 0,
@@ -939,7 +1041,7 @@ buffered_rmqueue(struct zone *zone, int

    if (page == NULL) {
        spin_lock_irqsave(&zone->lock, flags);
- page = __rmqueue(zone, order, gfp_flags);
+ page = __rmqueue(zone, order, gfp_flags, &flags);
        spin_unlock_irqrestore(&zone->lock, flags);
    }

@@ -948,9 +1050,6 @@ buffered_rmqueue(struct zone *zone, int
    mod_page_state_zone(zone, pgalloc, 1 << order);
    prep_new_page(page, order);

- if (gfp_flags & __GFP_ZERO)
- prep_zero_page(page, order, gfp_flags);
-
    if (order && (gfp_flags & __GFP_COMP))
        prep_compound_page(page, order);
}
@@ -1483,20 +1582,30 @@ void show_free_areas(void)
    printk("\n");

    for (cpu = 0; cpu < NR_CPUS; ++cpu) {
- struct per_cpu_pageset *pageset;
+ struct per_cpu_pageset *pageset, *pageset_zero;

        if (!cpu_possible(cpu))
            continue;

        pageset = zone->pageset + cpu;
+ pageset_zero = zone->pageset_zero + cpu;

- for (temperature = 0; temperature < 2; temperature++)
- printk("cpu %d %s: low %d, high %d, batch %d\n",
+ for (temperature = 0; temperature < 2; temperature++) {
+ printk("cpu %d general-page %s: low %d, high %d, batch %d\n",
            cpu,
            temperature ? "cold" : "hot",
            pageset->pcp[temperature].low,
            pageset->pcp[temperature].high,
            pageset->pcp[temperature].batch);
+
+ printk("cpu %d zero-page %s: low %d, high %d, batch %d\n",
+ cpu,
+ temperature ? "cold" : "hot",

```

Linux-Kernel: [PATCH] 2/2 Prezeroing large blocks of pages during allocation Version 4

```

+ pageset_zero->pcp[temperature].low,
+ pageset_zero->pcp[temperature].high,
+ pageset_zero->pcp[temperature].batch);
+
+ }
    }
}

@@ -1975,6 +2084,21 @@ static void __init free_area_init_core(s
    pcp->high = 2 * batch;
    pcp->batch = 1 * batch;
    INIT_LIST_HEAD(&pcp->list);
+
+ pcp = &zone->pageset_zero[cpu].pcp[0]; /* hot */
+ pcp->count = 0;
+ pcp->low = 2 * batch;
+ pcp->high = 6 * batch;
+ pcp->batch = 1 * batch;
+ INIT_LIST_HEAD(&pcp->list);
+
+ pcp = &zone->pageset_zero[cpu].pcp[1]; /* cold */
+ pcp->count = 0;
+ pcp->low = 0;
+ pcp->high = 2 * batch;
+ pcp->batch = 1 * batch;
+ INIT_LIST_HEAD(&pcp->list);
+
    }
    printk(KERN_DEBUG " %s zone: %lu pages, LIFO batch:%lu\n",
           zone_names[j], realsize, batch);
@@ -2029,6 +2153,8 @@ static void __init free_area_init_core(s
    sizeof(zone->bulk_requests));
    memset((unsigned long *)zone->bulk_allocated, 0,
           sizeof(zone->bulk_allocated));
+ memset((unsigned long *)zone->zeroblock_count, 0,
+ sizeof(zone->zeroblock_count));
    zone->kernnorclm_partial_steal=0;
    zone->kernnorclm_full_steal=0;
#endif
@@ -2133,12 +2259,14 @@ static int frag_show(struct seq_file *m,
    unsigned long alloc_count[ALLOC_TYPES];
    unsigned long bulk_requests[MAX_ORDER];
    unsigned long bulk_allocated[MAX_ORDER];
+ unsigned long zeroblock_count[MAX_ORDER];

    memset(reserve_count, 0, sizeof(reserve_count));
    memset(fallback_count, 0, sizeof(fallback_count));
    memset(alloc_count, 0, sizeof(alloc_count));
    memset(bulk_requests, 0, sizeof(bulk_requests));
    memset(bulk_allocated, 0, sizeof(bulk_allocated));
+ memset(zeroblock_count, 0, sizeof(zeroblock_count));

```

```

#endif

@@ -2207,18 +2335,25 @@ static int frag_show(struct seq_file *m,
    seq_printf(m, "Partial steal: %lu\n", zone->kernnorclm_partial_steal);
    seq_printf(m, "Full steal: %lu\n", zone->kernnorclm_full_steal);

- seq_printf(m, "Bulk requests ");
+ seq_printf(m, "Bulk requests ");
    for (i=0; i< MAX_ORDER; i++) {
        seq_printf(m, "%7lu ", zone->bulk_requests[i]);
        bulk_requests[i] += zone->bulk_requests[i];
    }
- seq_printf(m, "\nBulk allocated ");
+ seq_printf(m, "\nBulk allocated ");
    for (i=0; i< MAX_ORDER; i++) {
        seq_printf(m, "%7lu ", zone->bulk_allocated[i]);
        bulk_allocated[i] += zone->bulk_allocated[i];
    }
    seq_printf(m, "\n");

+ seq_printf(m, "Zeroblock count ");
+ for (i=0; i< MAX_ORDER; i++) {
+ seq_printf(m, "%7lu ", zone->zeroblock_count[i]);
+ zeroblock_count[i] += zone->zeroblock_count[i];
+ }
+ seq_printf(m, "\n");
+
    global_steal += zone->global_steal;
    global_refill += zone->global_refill;
    kernnorclm_partial_steal += zone->kernnorclm_partial_steal;
@@ -2234,7 +2369,7 @@ static int frag_show(struct seq_file *m,
    reserve_count[i] += zone->reserve_count[i];
    fallback_count[i] += zone->fallback_count[i];
}

-
+
#endif
    spin_unlock_irqrestore(&zone->lock, flags);
}

@@ -2246,16 +2381,23 @@ static int frag_show(struct seq_file *m,
    seq_printf(m, "Global refills: %lu\n", global_refill);
    seq_printf(m, "Partial steal: %lu\n", kernnorclm_partial_steal);
    seq_printf(m, "Full steal: %lu\n", kernnorclm_full_steal);
- seq_printf(m, "Bulk requests ");
+ seq_printf(m, "Bulk requests ");
    for (i=0; i< MAX_ORDER; i++) {
        seq_printf(m, "%7lu ", bulk_requests[i]);
    }
- seq_printf(m, "\nBulk allocated ");
+ seq_printf(m, "\nBulk allocated ");

```

Linux-Kernel: [PATCH] 2/2 Prezeroing large blocks of pages during allocation Version 4

```
for (i=0; i < MAX_ORDER; i++) {  
    seq_printf(m, "%7lu ", bulk_allocated[i]);  
}  
seq_printf(m, "\n");
```

```
+ seq_printf(m, "Zeroblock count ");  
+ for (i=0; i < MAX_ORDER; i++) {  
+ seq_printf(m, "%7lu ", zeroblock_count[i]);  
+ }  
+ seq_printf(m, "\n");  
+  
+
```

```
for (i=0; i < ALLOC_TYPES; i++) {  
    seq_printf(m, "%-10s Allocs: %-10lu Reserve: %-10lu Fallbacks: %-10lu\n",  
        type_names[i],
```

–
To unsubscribe from this list: send the line "unsubscribe linux-kernel" in
the body of a message to majordomo@vger.kernel.org
More majordomo info at <http://vger.kernel.org/majordomo-info.html>
Please read the FAQ at <http://www.tux.org/lkml/>