

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2008-05/msg05339.html>

- *From:* "Huang, Ying" <ying.huang@xxxxxxxx>
 - *Date:* Mon, 12 May 2008 14:40:41 +0800
-

This patch implements a prototype of kexec multi-stage load. With this patch, the "backup pages map" can be passed to kexeced kernel via /sbin/kexec; and the sys_kexec_load can be used to load large hibernated image with huge number of segments.

In kexec based hibernation, resuming from disk is implemented as loading the hibernated disk image with sys_kexec_load(). But unlike the normal kexec load, the hibernated image may have huge number of segments. So multi-stage loading is necessary for kexec load based resuming from disk implementation. And, multi-stage loading is also necessary for parameter passing from original kernel to kexeced kernel because some information such as "backup pages map" is not available before loading.

Four stages are defined:

- KS_start: start stage; begin a new kexec loading; there must be only one KS_start stage in one kexec loading.
- KS_mid: middle stage; continue load some segments; there may be many or zero KS_mid stages in one kexec loading; follows a KS_start or KS_mid stage.
- KS_final: final stage; finish a kexec loading; there must be only one KS_final stage in one kexec loading; follows a KS_start or KS_mid stage.
- KS_full: back compatible with original loading semantics, finish all work of a kexec loading in one KS_full stage.

Overlapping between pages of different segments is allowed to support "parameter passing".

During loading, a hash table mapped from destination page to source

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

page is used instead of original linear mapping implementation. Because the hibernated image may be very large (up to near the size of physical memory), it is very time-consuming to search a source page given the destination page, which is used to check whether a newly allocated page is in the range of allocated destination pages. The original mapping is only used by assembly code to swap the page contents. This map is also exported to user space via `/proc/kexec_pgmap`, so that `/sbin/kexec` can use it to construct the "backup pages map" parameter for kexeced kernel.

This patch is based on Linux kernel 2.6.25 and `kexec_jump` patch, and has been tested on an IBM T42.

Signed-off-by: Huang Ying <ying.huang@xxxxxxxx>

```
---
include/linux/kexec.h | 19 +
kernel/kexec.c | 608 ++++++-----
2 files changed, 478 insertions(+), 149 deletions(-)
```

```
--- a/kernel/kexec.c
+++ b/kernel/kexec.c
@@ -29,6 +29,10 @@
#include <linux/pm.h>
#include <linux/cpu.h>
#include <linux/console.h>
+#include <linux/proc_fs.h>
+#include <linux/seq_file.h>
+#include <linux/hash.h>
+#include <linux/pfn.h>

#include <asm/page.h>
#include <asm/uaccess.h>
@@ -107,43 +111,29 @@ int kexec_should_crash(struct task_struct
*/
#define KIMAGE_NO_DEST (-1UL)

+#define KIMAGE_HASH_BITS 10
+#define KIMAGE_PGTABLE_SIZE (1 << KIMAGE_HASH_BITS)
+
+struct pgmap {
+ struct hlist_node hlist;
+ unsigned long dst_pfn;
+ unsigned long src_pfn;
+};
+
static int kimage_is_destination_range(struct kimage *image,
unsigned long start, unsigned long end);
static struct page *kimage_alloc_page(struct kimage *image,
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
gfp_t gfp_mask,
unsigned long dest);

-static int do_kimage_alloc(struct kimage **rimage, unsigned long entry,
- unsigned long nr_segments,
- struct kexec_segment __user *segments)
+static int kimage_copy_segments(struct kimage *image,
+ unsigned long nr_segments,
+ struct kexec_segment __user *segments)
{
size_t segment_bytes;
- struct kimage *image;
unsigned long i;
int result;

- /* Allocate a controlling structure */
- result = -ENOMEM;
- image = kzalloc(sizeof(*image), GFP_KERNEL);
- if (!image)
- goto out;
-
- image->head = 0;
- image->entry = &image->head;
- image->last_entry = &image->head;
- image->control_page = ~0; /* By default this does not apply */
- image->start = entry;
- image->type = KEXEC_TYPE_DEFAULT;
-
- /* Initialize the list of control pages */
- INIT_LIST_HEAD(&image->control_pages);
-
- /* Initialize the list of destination pages */
- INIT_LIST_HEAD(&image->dest_pages);
-
- /* Initialize the list of unuseable pages */
- INIT_LIST_HEAD(&image->unuseable_pages);
-
/* Read in the segments */
image->nr_segments = nr_segments;
segment_bytes = nr_segments * sizeof(*segments);
@@ -210,6 +200,44 @@ static int do_kimage_alloc(struct kimage
}

result = 0;
+ out:
+ return result;
+}
+
+static int do_kimage_alloc(struct kimage **rimage, unsigned long entry,
+ unsigned long nr_segments,
+ struct kexec_segment __user *segments)
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
+{
+ struct kimage *image;
+ int result;
+
+ /* Allocate a controlling structure */
+ result = -ENOMEM;
+ image = kzalloc(sizeof(*image), GFP_KERNEL);
+ if (!image)
+ goto out;
+
+ image->head = 0;
+ image->entry = &image->head;
+ image->last_entry = &image->head;
+ image->control_page = ~0; /* By default this does not apply */
+ image->start = entry;
+ image->type = KEXEC_TYPE_DEFAULT;
+
+ /* Initialize the list of control pages */
+ INIT_LIST_HEAD(&image->control_pages);
+
+ /* Initialize the list of destination pages */
+ image->dest_pages = NULL;
+
+ /* Initialize the list of unuseable pages */
+ INIT_LIST_HEAD(&image->unuseable_pages);
+
+ result = kimage_copy_segments(image, nr_segments, segments);
+ if (result)
+ goto out;
+
+ result = 0;
out:
if (result == 0)
*rimage = image;
@@ -220,20 +248,9 @@ out:
}

```

```
-static int kimage_normal_alloc(struct kimage **rimage, unsigned long entry,
- unsigned long nr_segments,
- struct kexec_segment __user *segments)
+static int do_kimage_alloc_normal_control_pages(struct kimage *image)
{
int result;
- struct kimage *image;
-
- /* Allocate and initialize a controlling structure */
- image = NULL;
- result = do_kimage_alloc(&image, entry, nr_segments, segments);
- if (result)
- goto out;

```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
-
- *rimage = image;

/*
 * Find a location for the control code buffer, and add it
@@ -256,6 +273,32 @@ static int kimage_normal_alloc(struct ki

result = 0;
out:
+ return result;
+}
+
+static int kimage_normal_alloc(struct kimage **rimage, unsigned long entry,
+ unsigned long nr_segments,
+ struct kexec_segment __user *segments)
+{
+ int result;
+ struct kimage *image;
+
+ /* Allocate and initialize a controlling structure */
+ image = NULL;
+ result = do_kimage_alloc(&image, entry, nr_segments, segments);
+ if (result)
+ goto out;
+
+ *rimage = image;
+
+ result = -ENOMEM;
+ image->pgtable = kzalloc(sizeof(struct hlist_head) *
+ KIMAGE_PGTABLE_SIZE, GFP_KERNEL);
+ if (!image->pgtable)
+ goto out;
+
+ result = 0;
+ out:
if (result == 0)
*rimage = image;
else
@@ -333,9 +376,65 @@ out:
return result;
}

-static int kimage_is_destination_range(struct kimage *image,
- unsigned long start,
- unsigned long end)
+static struct pgmap *kimage_get_pgmap(struct kimage *image,
+ unsigned long dst_pfn)
+{
+ struct hlist_head *head;
+ struct hlist_node *node;
+ struct pgmap *map;
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
+
+ head = &image->pgtable[hash_long(dst_pfn, KIMAGE_HASH_BITS)];
+ hlist_for_each_entry(map, node, head, hlist) {
+ if (map->dst_pfn == dst_pfn)
+ return map;
+ }
+ return NULL;
+}
+
+static struct pgmap *kimage_add_pgmap(struct kimage *image,
+ unsigned long dst_pfn,
+ unsigned long src_pfn)
+{
+ struct pgmap *map;
+ struct hlist_head *head;
+
+ map = kzalloc(sizeof(struct pgmap), GFP_KERNEL);
+ if (!map)
+ return NULL;
+ map->dst_pfn = dst_pfn;
+ map->src_pfn = src_pfn;
+ head = &image->pgtable[hash_long(dst_pfn, KIMAGE_HASH_BITS)];
+ hlist_add_head(&map->hlist, head);
+ return map;
+}
+
+static void kimage_del_pgmap(struct kimage *image,
+ struct pgmap *map)
+{
+ hlist_del(&map->hlist);
+}
+
+#define for_each_pgmap(image, hhead, hnode, map) \
+ for (hhead = image->pgtable; \
+ hhead < image->pgtable + KIMAGE_PGTABLE_SIZE; \
+ hhead++) \
+ hlist_for_each_entry(map, hnode, hhead, hlist)
+
+static int kimage_is_old_destination_range(struct kimage *image,
+ unsigned long start,
+ unsigned long end)
+{
+ unsigned long pfn;
+
+ for (pfn = PFN_DOWN(start); pfn < PFN_UP(end); pfn++)
+ if (kimage_get_pgmap(image, pfn))
+ return 1;
+ return 0;
+}
+
+static int kimage_is_new_destination_range(struct kimage *image,
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
+ unsigned long start,
+ unsigned long end)
{
unsigned long i;

@@ -351,6 +450,14 @@ static int kimage_is_destination_range(s
return 0;
}

+static int kimage_is_destination_range(struct kimage *image,
+ unsigned long start,
+ unsigned long end)
+{
+ return kimage_is_new_destination_range(image, start, end) ||
+ kimage_is_old_destination_range(image, start, end);
+}
+
static struct page *kimage_alloc_pages(gfp_t gfp_mask, unsigned int order)
{
struct page *pages;
@@ -592,15 +699,46 @@ static int kimage_add_page(struct kimage
}

+static void kimage_add_dest_page(struct kimage *image, struct page *page)
+{
+ page->lru.next = image->dest_pages;
+ image->dest_pages = &page->lru;
+ page->lru.prev = NULL;
+}
+
+static void kimage_free_dest_pages(struct kimage *image)
+{
+ struct list_head *lh = image->dest_pages;
+ struct page *page;
+
+ while (lh) {
+ page = list_entry(lh, struct page, lru);
+ lh->prev = NULL;
+ lh = lh->next;
+ kimage_free_pages(page);
+ }
+ image->dest_pages = NULL;
+}
+
+static void kimage_mark_inplace_page(struct kimage *image, struct page *page)
+{
+ BUG_ON(page->lru.prev);
+ page->lru.prev = (struct list_head *)image;
+}
+
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
+static int kimage_is_inplace_page(struct kimage *image, struct page *page)
+{
+ return page->lru.prev == (struct list_head *)image;
+}
+
static void kimage_free_extra_pages(struct kimage *image)
{
- /* Walk through and free any extra destination pages I may have */
- kimage_free_page_list(&image->dest_pages);
+ kimage_free_dest_pages(image);

/* Walk through and free any unuseable pages I have cached */
kimage_free_page_list(&image->unuseable_pages);
-
}
+
static int kimage_terminate(struct kimage *image)
{
if (*image->entry != 0)
@@ -616,39 +754,64 @@ static int kimage_terminate(struct kimag
ptr = (entry & IND_INDIRECTION)? \
phys_to_virt((entry & PAGE_MASK)): ptr + 1)

- static void kimage_free_entry(kimage_entry_t entry)
+static kimage_entry_t *kimage_dst_in_swap_map(struct kimage *image,
+ unsigned long page)
{
- struct page *page;
+ kimage_entry_t *ptr, entry;
+ unsigned long destination = 0;
+
+ for_each_kimage_entry(image, ptr, entry) {
+ if (entry & IND_DESTINATION)
+ destination = entry & PAGE_MASK;
+ else if (entry & IND_SOURCE) {
+ if (page == destination)
+ return ptr;
+ destination += PAGE_SIZE;
+ }
+ }
+
+ return NULL;
+}

- page = pfn_to_page(entry >> PAGE_SHIFT);
- kimage_free_pages(page);
+static int kimage_create_swap_map(struct kimage *image)
+{
+ int result;
+ struct hlist_head *h;
+ struct hlist_node *n;
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
+ struct pgmap *map;
+
+ for_each_pgmap(image, h, n, map) {
+ result = kimage_set_destination(image, PFN_PHYS(map->dst_pfn));
+ if (result)
+ return result;
+ result = kimage_add_page(image, PFN_PHYS(map->src_pfn));
+ if (result)
+ return result;
+ }
+ kimage_terminate(image);
+ return 0;
}
```

```
static void kimage_free(struct kimage *image)
{
- kimage_entry_t *ptr, entry;
- kimage_entry_t ind = 0;
+ struct hlist_head *hhead;
+ struct hlist_node *hnode, *hnode_tmp;
+ struct pgmap *map;
```

```
if (!image)
return;
```

```
kimage_free_extra_pages(image);
- for_each_kimage_entry(image, ptr, entry) {
- if (entry & IND_INDIRECTION) {
- /* Free the previous indirection page */
- if (ind & IND_INDIRECTION)
- kimage_free_entry(ind);
- /* Save this indirection page until we are
- * done with it.
- */
- ind = entry;
+
+ for (hhead = image->pgtable;
+ hhead < image->pgtable + KIMAGE_PGTABLE_SIZE;
+ hhead++)
+ hlist_for_each_entry_safe(map, hnode, hnode_tmp,
+ hhead, hlist) {
+ hlist_del(hnode);
+ kfree(map);
+ }
- else if (entry & IND_SOURCE)
- kimage_free_entry(entry);
- }
- /* Free the final indirection page */
- if (ind & IND_INDIRECTION)
- kimage_free_entry(ind);
+ kfree(image->pgtable);
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
/* Handle any machine specific cleanup */
machine_kexec_cleanup(image);
@@ -658,25 +821,6 @@ static void kimage_free(struct kimage *i
kfree(image);
}

-static kimage_entry_t *kimage_dst_used(struct kimage *image,
- unsigned long page)
- {
- kimage_entry_t *ptr, entry;
- unsigned long destination = 0;
-
- for_each_kimage_entry(image, ptr, entry) {
- if (entry & IND_DESTINATION)
- destination = entry & PAGE_MASK;
- else if (entry & IND_SOURCE) {
- if (page == destination)
- return ptr;
- destination += PAGE_SIZE;
- }
- }
-
- return NULL;
- }
-
static struct page *kimage_alloc_page(struct kimage *image,
gfp_t gfp_mask,
unsigned long destination)
@@ -700,23 +844,19 @@ static struct page *kimage_alloc_page(st
* be fixed.
*/
struct page *page;
- unsigned long addr;
+ unsigned long addr, dst_pfn;
+ struct pgmap *map;
+
+ dst_pfn = PFN_DOWN(destination);
+ page = pfn_to_page(dst_pfn);
+ if (kimage_is_inplace_page(image, page))
+ return page;
+ map = kimage_get_pgmap(image, dst_pfn);
+ if (map)
+ return pfn_to_page(map->src_pfn);

- /*
- * Walk through the list of destination pages, and see if I
- * have a match.
- */
- list_for_each_entry(page, &image->dest_pages, lru) {
- addr = page_to_pfn(page) << PAGE_SHIFT;
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
- if (addr == destination) {
- list_del(&page->lru);
- return page;
- }
- }
page = NULL;
while (1) {
- kimage_entry_t *old;
-
/* Allocate a page, if we run out of memory give up */
page = kimage_alloc_pages(gfp_mask, 0);
if (!page)
@@ -727,51 +867,94 @@ static struct page *kimage_alloc_page(st
list_add(&page->lru, &image->unuseable_pages);
continue;
}
+ kimage_add_dest_page(image, page);
addr = page_to_pfn(page) << PAGE_SHIFT;

/* If it is the destination page we want use it */
- if (addr == destination)
- break;
-
- /* If the page is not a destination page use it */
- if (!kimage_is_destination_range(image, addr,
- addr + PAGE_SIZE))
- break;
+ if (addr == destination) {
+ kimage_mark_inplace_page(image, page);
+ return page;
+ }

+ map = kimage_get_pgmap(image, PFN_DOWN(addr));
/*
* I know that the page is someones destination page.
* See if there is already a source page for this
* destination page. And if so swap the source pages.
*/
- old = kimage_dst_used(image, addr);
- if (old) {
- /* If so move it */
- unsigned long old_addr;
- struct page *old_page;
-
- old_addr = *old & PAGE_MASK;
- old_page = pfn_to_page(old_addr >> PAGE_SHIFT);
- copy_highpage(page, old_page);
- *old = addr | (*old & ~PAGE_MASK);
-
- /* The old page I have found cannot be a
- * destination page, so return it.
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
- */
- addr = old_addr;
- page = old_page;
- break;
+ if (map) {
+ kimage_entry_t *old;
+ struct page *src_page;
+ unsigned long src_pfn;
+
+ src_pfn = map->src_pfn;
+ src_page = pfn_to_page(src_pfn);
+ copy_highpage(page, src_page);
+ kimage_del_pgmap(image, map);
+ kfree(map);
+ kimage_mark_inplace_page(image, page);
+
+ old = kimage_dst_in_swap_map(image, addr);
+ if (old)
+ *old = addr | (*old & ~PAGE_MASK);
+
+ addr = PFN_PHYS(src_pfn);
+ page = src_page;
+ }
+
+ if (kimage_is_new_destination_range(image, addr,
+ addr + PAGE_SIZE)) {
+ kimage_mark_inplace_page(image, page);
+ continue;
+ }
- else {
- /* Place the page on the destination list I
- * will use it later.
- */
- list_add(&page->lru, &image->dest_pages);
+
+ if (destination != KIMAGE_NO_DEST) {
+ map = kimage_add_pgmap(image, dst_pfn, PFN_DOWN(addr));
+ if (!map) {
+ kimage_free_pages(page);
+ return NULL;
+ }
+ }
+ break;
+ }

return page;
}

+static int fix_old_pages(struct kimage *image)
+{
+ struct hlist_head *hhead;
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
+ struct hlist_node **pprev;
+ struct pgmap *map;
+ unsigned long addr;
+ struct page *page, *src_page;
+
+ for (hhead = image->pgtable;
+ hhead < image->pgtable + KIMAGE_PGTABLE_SIZE;
+ hhead++) {
+ pprev = &hhead->first;
+ while (*pprev) {
+ map = hlist_entry(*pprev, struct pgmap, hlist);
+ addr = PFN_PHYS(map->src_pfn);
+ if (!kimage_is_new_destination_range(image, addr,
+ addr+PAGE_SIZE)) {
+ pprev = &(map->hlist.next);
+ continue;
+ }
+ kimage_del_pgmap(image, map);
+ page = kimage_alloc_page(image, GFP_KERNEL,
+ PFN_PHYS(map->dst_pfn));
+ if (!page)
+ return -ENOMEM;
+ src_page = pfn_to_page(map->src_pfn);
+ copy_highpage(page, src_page);
+ kfree(map);
+ kimage_mark_inplace_page(image, src_page);
+ }
+ }
+ return 0;
+}
+
static int kimage_load_normal_segment(struct kimage *image,
struct kexec_segment *segment)
{
@@ -786,10 +969,6 @@ static int kimage_load_normal_segment(st
mbytes = segment->memsz;
maddr = segment->mem;

- result = kimage_set_destination(image, maddr);
- if (result < 0)
- goto out;
-
while (mbytes) {
struct page *page;
char *ptr;
@@ -800,10 +979,6 @@ static int kimage_load_normal_segment(st
result = -ENOMEM;
goto out;
}
- result = kimage_add_page(image, page_to_pfn(page)
- << PAGE_SHIFT);
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
- if (result < 0)
- goto out;

ptr = kmap(page);
/* Start with a clear page */
@@ -940,6 +1115,7 @@ asmlinkage long sys_kexec_load(unsigned
struct kimage **dest_image, *image;
int locked;
int result;
+ int stage;

/* We only trust the superuser with rebooting the system. */
if (!capable(CAP_SYS_BOOT))
@@ -978,6 +1154,23 @@ asmlinkage long sys_kexec_load(unsigned
if (locked)
return -EBUSY;

+ stage = KEXEC_GET_STAGE(flags);
+
+ /* multi-stage load is not supported for kcrash */
+ if ((flags & KEXEC_ON_CRASH) && stage != KS_full) {
+ result = -EINVAL;
+ goto out;
+ }
+ if (!kexec_image && stage != KS_full && stage != KS_start) {
+ result = -EINVAL;
+ goto out;
+ }
+ if (kexec_image && kexec_image->stage == KS_final &&
+ stage != KS_start && stage != KS_full) {
+ result = -EINVAL;
+ goto out;
+ }
+
dest_image = &kexec_image;
if (flags & KEXEC_ON_CRASH)
dest_image = &kexec_crash_image;
@@ -985,11 +1178,20 @@ asmlinkage long sys_kexec_load(unsigned
unsigned long i;

/* Loading another kernel to reboot into */
- if ((flags & KEXEC_ON_CRASH) == 0)
- result = kimage_normal_alloc(&image, entry,
- nr_segments, segments);
+ if ((flags & KEXEC_ON_CRASH) == 0) {
+ if (stage == KS_full || stage == KS_start)
+ result = kimage_normal_alloc(&image,
+ entry,
+ nr_segments,
+ segments);
+ else {
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
+ image = kexec_image;
+ result = kimage_copy_segments(image,
+ nr_segments,
+ segments);
+ }
/* Loading another kernel to switch to if this one crashes */
- else if (flags & KEXEC_ON_CRASH) {
+ } else if (flags & KEXEC_ON_CRASH) {
/* Free any current crash dump kernel before
* we corrupt it.
*/
@@ -1002,7 +1204,9 @@ asmlinkage long sys_kexec_load(unsigned

if (flags & KEXEC_PRESERVE_CONTEXT)
image->preserve_context = 1;
- result = machine_kexec_prepare(image);
+ image->stage = stage;
+
+ result = fix_old_pages(image);
if (result)
goto out;

@@ -1011,17 +1215,32 @@ asmlinkage long sys_kexec_load(unsigned
if (result)
goto out;
}
- result = kimage_terminate(image);
- if (result)
- goto out;
- }
- /* Install the new kernel, and Uninstall the old */
- image = xchg(dest_image, image);

+ if (stage == KS_full || stage == KS_final) {
+ if ((flags & KEXEC_ON_CRASH) == 0) {
+ result = do_kimage_alloc_normal_control_pages(
+ image);
+ if (result)
+ goto out;
+ }
+ result = machine_kexec_prepare(image);
+ if (result)
+ goto out;
+ result = kimage_create_swap_map(image);
+ if (result)
+ goto out;
+ }
+ }
+ if (stage == KS_full || stage == KS_start) {
+ /* Install the new kernel, and Uninstall the old */
+ image = xchg(dest_image, image);
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
+ } else
+ image = NULL;
out:
locked = xchg(&kexec_lock, 0); /* Release the mutex */
BUG_ON(!locked);
- kimage_free(image);
+ if (image)
+ kimage_free(image);

return result;
}
@@ -1065,6 +1284,98 @@ asmlinkage long compat_sys_kexec_load(un
}
#endif

+struct kexec_pgmap_pos
+{
+ struct hlist_head *hhead;
+ struct hlist_head *hhead_end;
+ struct pgmap *map;
+};
+
+static int kexec_pgmap_show(struct seq_file *f, void *v)
+{
+ struct kexec_pgmap_pos *ptr = v;
+
+ seq_printf(f, "0x%016lx\t0x%016lx\n",
+ ptr->map->dst_pfn, ptr->map->src_pfn);
+
+ return 0;
+}
+
+static void *kexec_pgmap_start(struct seq_file *f, loff_t *ppos)
+{
+ struct kexec_pgmap_pos *ptr;
+ struct hlist_node *hnode;
+ loff_t pos = *ppos;
+
+ ptr = kmalloc(sizeof(*ptr), GFP_KERNEL);
+ ptr->hhead_end = kexec_image->pgtable + KIMAGE_PGTABLE_SIZE;
+ for_each_pgmap(kexec_image, ptr->hhead, hnode, ptr->map) {
+ if (pos-- == 0)
+ return ptr;
+ }
+ kfree(ptr);
+ return NULL;
+}
+
+static void *kexec_pgmap_next(struct seq_file *f, void *v, loff_t *ppos)
+{
+ struct kexec_pgmap_pos *ptr = v;
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
+ struct hlist_node *hnode = ptr->map->hlist.next;
+
+ if (hnode) {
+ ptr->map = hlist_entry(hnode, struct pgmap, hlist);
+ (*ppos)++;
+ return ptr;
+ }
+ for (ptr->hhead++; ptr->hhead < ptr->hhead_end; ptr->hhead++) {
+ hlist_for_each_entry(ptr->map, hnode, ptr->hhead, hlist) {
+ (*ppos)++;
+ return ptr;
+ }
+ }
+ return NULL;
+}
+
+static void kexec_pgmap_stop(struct seq_file *f, void *v)
+{
+ kfree(v);
+}
+
+static const struct seq_operations kexec_pgmap_ops = {
+ .start = kexec_pgmap_start,
+ .next = kexec_pgmap_next,
+ .stop = kexec_pgmap_stop,
+ .show = kexec_pgmap_show
+};
+
+static int kexec_pgmap_open(struct inode *inode, struct file *file)
+{
+ if (kexec_image)
+ return seq_open(file, &kexec_pgmap_ops);
+ else
+ return -ENODEV;
+}
+
+static const struct file_operations kexec_pgmap_file_ops = {
+ .open = kexec_pgmap_open,
+ .read = seq_read,
+ .llseek = seq_lseek,
+ .release = seq_release,
+};
+
+static int __init kexec_pgmap_file_init(void)
+{
+ struct proc_dir_entry *entry;
+
+ entry = proc_create("kexec_pgmap", S_IRUSR, NULL,
+ &kexec_pgmap_file_ops);
+ if (!entry)
+ return -ENOMEM;
+}
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
+ return 0;
+}
+
+late_initcall(kexec_pgmap_file_init);
+
void crash_kexec(struct pt_regs *regs)
{
int locked;
@@ -1431,6 +1742,8 @@ int kexec_jump(struct kimage *image)
{
int error = 0;

+ if (image->stage != KS_full && image->stage != KS_final)
+ return -EINVAL;
mutex_lock(&pm_mutex);
if (image->preserve_context) {
pm_prepare_console();
@@ -1458,6 +1771,7 @@ int kexec_jump(struct kimage *image)
if (error)
goto Enable_irqs;
save_processor_state();
+ memcpy(vmcoreinfo_note, vmcoreinfo_data, vmcoreinfo_size);
}
machine_kexec(image);

--- a/include/linux/kexec.h
+++ b/include/linux/kexec.h
@@ -46,6 +46,13 @@
KEXEC_CORE_NOTE_NAME_BYTES + \
KEXEC_CORE_NOTE_DESC_BYTES )

+enum kexec_stage {
+ KS_full, /* including all following 3 stages */
+ KS_start,
+ KS_mid,
+ KS_final,
+};
+
+/*
+ * This structure is used to hold the arguments that are used when loading
+ * kernel binaries.
+@@ -89,8 +96,10 @@ struct kimage {
struct kexec_segment segment[KEXEC_SEGMENT_MAX];

struct list_head control_pages;
- struct list_head dest_pages;
struct list_head unuseable_pages;
+ struct list_head *dest_pages;
+
+ struct hlist_head *pgtable;
```

[PATCH] kexec based hibernation: a prototype of kexec multi-stage load

```
/* Address of next control page to allocate for crash kernels. */
unsigned long control_page;
@@ -100,6 +109,7 @@ struct kimage {
#define KEXEC_TYPE_DEFAULT 0
#define KEXEC_TYPE_CRASH 1
unsigned int preserve_context : 1;
+ unsigned int stage : 2;
};

@@ -160,8 +170,11 @@ extern int kexec_lock;
#define kexec_flush_icache_page(page)
#endif

+#define KEXEC_GET_STAGE(flags) (((flags)&KEXEC_STAGE_MASK)>>2)
+
#define KEXEC_ON_CRASH 0x00000001
#define KEXEC_PRESERVE_CONTEXT 0x00000002
+#define KEXEC_STAGE_MASK 0x0000000c
#define KEXEC_ARCH_MASK 0xffff0000

/* These values match the ELF architecture values.
@@ -180,7 +193,9 @@ extern int kexec_lock;
#define KEXEC_ARCH_MIPS ( 8 << 16)

/* List of defined/legal kexec flags */
-#define KEXEC_FLAGS (KEXEC_ON_CRASH | KEXEC_PRESERVE_CONTEXT)
+#define KEXEC_FLAGS (KEXEC_ON_CRASH | \
+ KEXEC_PRESERVE_CONTEXT | \
+ KEXEC_STAGE_MASK)

#define VMCOREINFO_BYTES (4096)
#define VMCOREINFO_NOTE_NAME "VMCOREINFO"
```

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