

[patch – v3] epoll ready set loops diet ...

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2007-02/msg10150.html>

- *From:* Davide Libenzi <davide1@xxxxxxxxxxxxxxxxxx>
 - *Date:* Wed, 28 Feb 2007 10:37:35 -0800 (PST)
-

ChangeLog:

v3) Removed the "revents" field from the epoll item structure, as suggested by Eric Dumazet

v2) In v1, I was trying to avoid to get the spinlock twice WRT yesterday patch, but it turns out I can't since the ready list will be travelling through a path w/out the ep->sem held. Oh well...

Epoll is doing multiple passes over the ready set at the moment, because of the constraints over the f_op->poll() call. Looking at the code again, I noticed that we already hold the epoll semaphore in read, and this (together with other locking conditions that hold while doing an epoll_wait()) can lead to a smarter way [1] to "ship" events to userspace (in a single pass).

This is a stress application that can be used to test the new code. It spawns multiple thread and call epoll_wait() and epoll_ctl() from many threads. Stress tested on my dual Opteron 254 w/out any problems.

<http://www.xmailserver.org/totalmess.c>

This is not a benchmark, just something that tries to stress and exploit possible problems with the new code.

Also, I made a stupid micro-benchmark:

<http://www.xmailserver.org/epwbench.c>

[1] Considering that epoll must be thread-safe, there are five ways we can be hit during an epoll_wait() transfer loop (ep_send_events()):

1) The epoll fd going away and calling ep_free

This just can't happen, since we did an fget() in sys_epoll_wait

2) An epoll_ctl(Epoll_CTL_DEL)

This can't happen because epoll_ctl() gets ep->sem in write, and

we're holding it in read during ep_send_events()

3) An fd stored inside the epoll fd going away

This can't happen because in eventpoll_release_file() we get ep->sem in write, and we're holding it in read during ep_send_events()

4) Another epoll_wait() happening on another thread

They both can be inside ep_send_events() at the same time, we get (splice) the ready-list under the spinlock, so each one will get its own ready list. Note that an fd cannot be at the same time inside more than one ready list, because ep_poll_callback() will not re-queue it if it sees it already linked:

```
if (ep_is_linked(&epi->rdllink))
    goto is_linked;
```

Another case that can happen, is two concurrent epoll_wait(), coming in with a userspace event buffer of size, say, ten. Suppose there are 50 event ready in the list. The first epoll_wait() will "steal" the whole list, while the second, seeing no events, will go to sleep. But at the end of ep_send_events() in the first epoll_wait(), we will re-inject surplus ready fds, and we will trigger the proper wake_up to the second epoll_wait().

5) ep_poll_callback() hitting us asynchronously

This is the tricky part. As I said above, the ep_is_linked() test done inside ep_poll_callback(), will guarantee us that until the item will result linked to a list, ep_poll_callback() will not try to re-queue it again (read, write data on any of its members). When we do a list_del() in ep_send_events(), the item will still satisfy the ep_is_linked() test (whatever data is written in prev/next, it'll never be its own pointer), so ep_poll_callback() will still leave us alone. It's only after the eventual smp_mb()+INIT_LIST_HEAD(&epi->rdllink) that it'll become visible to ep_poll_callback(), but at the point we're already past it.

Signed-off-by: Davide Libenzi <davidel@xxxxxxxxxxxxxxxx>

– Davide

eventpoll.c | 230 ++++++-----
1 file changed, 85 insertions(+), 145 deletions(-)

```
diff -Nru linux-2.6.20/fs/eventpoll.c linux-2.6.20.mod/fs/eventpoll.c
--- linux-2.6.20/fs/eventpoll.c 2007-02-04 10:44:54.000000000 -0800
+++ linux-2.6.20.mod/fs/eventpoll.c 2007-02-28 10:30:49.000000000 -0800
@@ -185,7 +185,7 @@

/*
 * Each file descriptor added to the eventpoll interface will
 - * have an entry of this type linked to the hash.
 + * have an entry of this type linked to the "rbr" RB tree.
 */
struct epitem {
/* RB-Tree node used to link this structure to the eventpoll rb-tree */
@@ -217,15 +217,6 @@

/* List header used to link this item to the "struct file" items list */
struct list_head flink;
-
- /* List header used to link the item to the transfer list */
- struct list_head txlink;
-
- /*
- * This is used during the collection/transfer of events to userspace
- * to pin items empty events set.
- */
- unsigned int revents;
};

/* Wrapper struct used by poll queueing */
@@ -258,11 +249,8 @@
static int ep_poll_callback(wait_queue_t *wait, unsigned mode, int sync, void *key);
static int ep_eventpoll_close(struct inode *inode, struct file *file);
static unsigned int ep_eventpoll_poll(struct file *file, poll_table *wait);
- static int ep_collect_ready_items(struct eventpoll *ep,
- struct list_head *txlist, int maxevents);
static int ep_send_events(struct eventpoll *ep, struct list_head *txlist,
- struct epoll_event __user *events);
- static void ep_reinject_items(struct eventpoll *ep, struct list_head *txlist);
+ struct epoll_event __user *events, int maxevents);
static int ep_events_transfer(struct eventpoll *ep,
struct epoll_event __user *events,
int maxevents);
@@ -355,17 +343,6 @@
return rb_parent(n) != n;
}

- /*
- * Remove the item from the list and perform its initialization.
- * This is useful for us because we can test if the item is linked
- * using "ep_is_linked(p)".

```

```

- */
-static inline void ep_list_del(struct list_head *p)
-{
- list_del(p);
- INIT_LIST_HEAD(p);
-}
-
/* Tells us if the item is currently linked */
static inline int ep_is_linked(struct list_head *p)
{
@@ -385,7 +362,7 @@
}

/* Tells if the epoll_ctl(2) operation needs an event copy from userspace */
-static inline int ep_op_hash_event(int op)
+static inline int ep_op_has_event(int op)
{
return op != EPOLL_CTL_DEL;
}
@@ -480,7 +457,7 @@
epi = list_entry(lsthead->next, struct epitem, flink);

ep = epi->ep;
- ep_list_del(&epi->flink);
+ list_del_init(&epi->flink);
down_write(&ep->sem);
ep_remove(ep, epi);
up_write(&ep->sem);
@@ -557,7 +534,7 @@
current, epfd, op, fd, event));

error = -EFAULT;
- if (ep_op_hash_event(op) &&
+ if (ep_op_has_event(op) &&
copy_from_user(&epds, event, sizeof(struct epoll_event)))
goto eexit_1;

@@ -594,7 +571,7 @@

down_write(&ep->sem);

- /* Try to lookup the file inside our hash table */
+ /* Try to lookup the file inside our RB tree */
epi = ep_find(ep, tfile, fd);

error = -EINVAL;
@@ -876,7 +853,7 @@
}

/*
- * Walks through the whole hash by freeing each "struct epitem". At this

```

[patch – v3] epoll ready set loops diet ...

```
+ * Walks through the whole tree by freeing each "struct epitem". At this
* point we are sure no poll callbacks will be lingering around, and also by
* write-holding "sem" we can be sure that no file cleanup code will hit
* us during this operation. So we can avoid the lock on "ep->lock".
@@ -891,7 +868,7 @@
```

```
/*
- * Search the file inside the eventpoll hash. It add usage count to
+ * Search the file inside the eventpoll tree. It add usage count to
* the returned item, so the caller must call ep_release_epitem()
* after finished using the "struct epitem".
*/
```

```
@@ -1011,7 +988,6 @@
ep_rb_initnode(&epi->rbn);
INIT_LIST_HEAD(&epi->rdllink);
INIT_LIST_HEAD(&epi->flink);
- INIT_LIST_HEAD(&epi->txlink);
INIT_LIST_HEAD(&epi->pwqlist);
epi->ep = ep;
ep_set_ffd(&epi->ffd, tfile, fd);
@@ -1080,7 +1056,7 @@
*/
write_lock_irqsave(&ep->lock, flags);
if (ep_is_linked(&epi->rdllink))
- ep_list_del(&epi->rdllink);
+ list_del_init(&epi->rdllink);
write_unlock_irqrestore(&ep->lock, flags);
```

```
kmem_cache_free(epi_cache, epi);
@@ -1119,7 +1095,7 @@
epi->event.data = event->data;
```

```
/*
- * If the item is not linked to the hash it means that it's on its
+ * If the item is not linked to the RB tree it means that it's on its
* way toward the removal. Do nothing in this case.
*/
```

```
if (ep_rb_linked(&epi->rbn)) {
@@ -1170,7 +1146,7 @@
while (!list_empty(lsthead)) {
pwq = list_entry(lsthead->next, struct eppoll_entry, llink);
```

```
- ep_list_del(&pwq->llink);
+ list_del_init(&pwq->llink);
remove_wait_queue(pwq->whead, &pwq->wait);
kmem_cache_free(pwq_cache, pwq);
}
```

```
@@ -1213,7 +1189,7 @@
* we want to remove it from this list to avoid stale events.
*/
```

```
if (ep_is_linked(&epi->rdllink))
- ep_list_del(&epi->rdllink);
+ list_del_init(&epi->rdllink);

error = 0;
eexit_1:
@@ -1226,7 +1202,7 @@

/*
- * Removes a "struct epitem" from the eventpoll hash and deallocates
+ * Removes a "struct epitem" from the eventpoll RB tree and deallocates
* all the associated resources.
*/
static int ep_remove(struct eventpoll *ep, struct epitem *epi)
@@ -1248,13 +1224,13 @@
/* Remove the current item from the list of epoll hooks */
spin_lock(&file->f_ep_lock);
if (ep_is_linked(&epi->flink))
- ep_list_del(&epi->flink);
+ list_del_init(&epi->flink);
spin_unlock(&file->f_ep_lock);

/* We need to acquire the write IRQ lock before calling ep_unlink() */
write_lock_irqsave(&ep->lock, flags);

- /* Really unlink the item from the hash */
+ /* Really unlink the item from the RB tree */
error = ep_unlink(ep, epi);

write_unlock_irqrestore(&ep->lock, flags);
@@ -1362,71 +1338,29 @@

/*
- * Since we have to release the lock during the __copy_to_user() operation and
- * during the f_op->poll() call, we try to collect the maximum number of items
- * by reducing the irqlock/irqunlock switching rate.
- */
-static int ep_collect_ready_items(struct eventpoll *ep, struct list_head *txlist, int maxevents)
- {
- int nepi;
- unsigned long flags;
- struct list_head *lsthead = &ep->rdllist, *lnk;
- struct epitem *epi;
-
- write_lock_irqsave(&ep->lock, flags);
-
- for (nepi = 0, lnk = lsthead->next; lnk != lsthead && nepi < maxevents;) {
- epi = list_entry(lnk, struct epitem, rdllink);
-

```

```

– lnk = lnk->next;
–
– /* If this file is already in the ready list we exit soon */
– if (!ep_is_linked(&epi->txlink)) {
– /*
– * This is initialized in this way so that the default
– * behaviour of the reinjecting code will be to push back
– * the item inside the ready list.
– */
– epi->revents = epi->event.events;
–
– /* Link the ready item into the transfer list */
– list_add(&epi->txlink, txlist);
– nepi++;
–
– /*
– * Unlink the item from the ready list.
– */
– ep_list_del(&epi->rdllink);
– }
– }
–
– write_unlock_irqrestore(&ep->lock, flags);
–
– return nepi;
–}
–
– /*
– * This function is called without holding the "ep->lock" since the call to
– * __copy_to_user() might sleep, and also f_op->poll() might reenale the IRQ
– * because of the way poll() is traditionally implemented in Linux.
– */
static int ep_send_events(struct eventpoll *ep, struct list_head *txlist,
– struct epoll_event __user *events)
+ struct epoll_event __user *events, int maxevents)
{
– int eventcnt = 0;
+ int eventcnt, error = -EFAULT, pwake = 0;
unsigned int revents;
– struct list_head *lnk;
+ unsigned long flags;
struct epitem *epi;
+ struct list_head injlist;
+
+ INIT_LIST_HEAD(&injlist);

/*
– * We can loop without lock because this is a task private list.
– * The test done during the collection loop will guarantee us that
– * another task will not try to collect this file. Also, items

```

[patch – v3] epoll ready set loops diet ...

```
– * cannot vanish during the loop because we are holding "sem".
+ * We just splice'd out the ep->rdllist in ep_collect_ready_items().
+ * Items cannot vanish during the loop because we are holding "sem" in read.
*/
– list_for_each(lnk, txlist) {
– epi = list_entry(lnk, struct epitem, txlink);
+ for (eventcnt = 0; !list_empty(txlist) && eventcnt < maxevents;) {
+ epi = list_entry(txlist->next, struct epitem, rdllink);
+ prefetch(epi->rdllink.next);

/*
* Get the ready file event set. We can safely use the file
@@ -1434,64 +1368,64 @@
* guarantee that both the file and the item will not vanish.
*/
revents = epi->ffd.file->f_op->poll(epi->ffd.file, NULL);
+ revents &= epi->event.events;

/*
– * Set the return event set for the current file descriptor.
– * Note that only the task task was successfully able to link
– * the item to its "txlist" will write this field.
+ * Is the event mask intersect the caller-requested one,
+ * deliver the event to userspace. Again, we are holding
+ * "sem" in read, so no operations coming from userspace
+ * can change the item.
*/
– epi->revents = revents & epi->event.events;
–
– if (epi->revents) {
– if (__put_user(epi->revents,
+ if (revents) {
+ if (__put_user(revents,
&events[eventcnt].events) ||
__put_user(epi->event.data,
&events[eventcnt].data))
– return -EFAULT;
+ goto errxit;
if (epi->event.events & EPOLLONESHOT)
epi->event.events &= EP_PRIVATE_BITS;
eventcnt++;
}
– }
– return eventcnt;
–}
–
–
–/*
– * Walk through the transfer list we collected with ep_collect_ready_items()
– * and, if 1) the item is still "alive" 2) its event set is not empty 3) it's
– * not already linked, links it to the ready list. Same as above, we are holding
```

[patch – v3] epoll ready set loops diet ...

```
– * "sem" so items cannot vanish underneath our nose.
– */
–static void ep_reinject_items(struct eventpoll *ep, struct list_head *txlist)
–{
– int ricnt = 0, pwake = 0;
– unsigned long flags;
– struct epitem *epi;
–
– write_lock_irqsave(&ep->lock, flags);
–
– while (!list_empty(txlist)) {
– epi = list_entry(txlist->next, struct epitem, txlink);
–
– /* Unlink the current item from the transfer list */
– ep_list_del(&epi->txlink);
–
– /*
– * If the item is no more linked to the interest set, we don't
– * have to push it inside the ready list because the following
– * ep_release_epitem() is going to drop it. Also, if the current
– * item is set to have an Edge Triggered behaviour, we don't have
– * to push it back either.
– * This is tricky. We are holding the "sem" in read, and this
– * means that the operations that can change the "linked" status
– * of the epoll item (epi->rbn and epi->rdllink), cannot touch them.
– * Also, since we are "linked" from a epi->rdllink POV (the item
– * is linked to our transmission list, we just splice'd), the
– * ep_poll_callback() cannot touch us either, because of the check
– * present in there. Another parallel epoll_wait() will not
– * get the same result set, since we spliced the ready list before.
– * Note that list_del() still shows the item as linked to the test
– * in ep_poll_callback().
– */
– if (ep_rb_linked(&epi->rbn) && !(epi->event.events & EPOLLET) &&
– (epi->revents & epi->event.events) && !ep_is_linked(&epi->rdllink)) {
– list_add_tail(&epi->rdllink, &ep->rdllist);
– ricnt++;
– }
+ list_del(&epi->rdllink);
+ if (!(epi->event.events & EPOLLET) && (revents & epi->event.events))
+ list_add_tail(&epi->rdllink, &injlist);
+ else {
+ /*
+ * Be sure the item is totally detached before re-init the
+ * list_head. After INIT_LIST_HEAD() is committed, the
+ * ep_poll_callback() can requeue the item again, but we
+ * don't care since we are already past it.
+ */
+ smp_mb();
+ INIT_LIST_HEAD(&epi->rdllink);
+ }
```

```

}
+ error = 0;

- if (ricnt) {
+ errxit:
+
+ /*
+ * If the re-injection list or the txlist are not empty, re-splice
+ * them to the ready list and do proper wakeups.
+ */
+ if (!list_empty(&injl) || !list_empty(txlist)) {
+ write_lock_irqsave(&ep->lock, flags);
+
+ list_splice(txlist, &ep->rdllist);
+ list_splice(&injl, &ep->rdllist);
+ /*
+ * Wake up ( if active ) both the eventpoll wait list and the ->poll()
+ * wait list.
+ @@ -1501,13 +1435,15 @@
+ TASK_INTERRUPTIBLE);
+ if (waitqueue_active(&ep->poll_wait))
+ pwake++;
- }

- write_unlock_irqrestore(&ep->lock, flags);
+ write_unlock_irqrestore(&ep->lock, flags);
+ }

+ /* We have to call this outside the lock */
+ if (pwake)
+ ep_poll_safewake(&psw, &ep->poll_wait);
+
+ return eventcnt == 0 ? error: eventcnt;
+ }

+ @@ -1517,7 +1453,8 @@
+ static int ep_events_transfer(struct eventpoll *ep,
+ struct epoll_event __user *events, int maxevents)
+ {
+ int eventcnt = 0;
+ int eventcnt;
+ unsigned long flags;
+ struct list_head txlist;

+ INIT_LIST_HEAD(&txlist);
+ @@ -1528,14 +1465,17 @@
+ */
+ down_read(&ep->sem);

- /* Collect/extract ready items */

```

[patch – v3] epoll ready set loops diet ...

```
– if (ep_collect_ready_items(ep, &txlist, maxevents) > 0) {
– /* Build result set in userspace */
– eventcnt = ep_send_events(ep, &txlist, events);
+ /*
+ * Steal the ready list, and re-init the original one to the
+ * empty list.
+ */
+ write_lock_irqsave(&ep->lock, flags);
+ list_splice(&ep->rdllist, &txlist);
+ INIT_LIST_HEAD(&ep->rdllist);
+ write_unlock_irqrestore(&ep->lock, flags);

– /* Reinject ready items into the ready list */
– ep_reinject_items(ep, &txlist);
– }
+ /* Build result set in userspace */
+ eventcnt = ep_send_events(ep, &txlist, events, maxevents);

up_read(&ep->sem);
```

–
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/>