Subject: [PATCH] usbatm: Update to use the kthread api.
Posted by ebiederm on Tue, 12 Dec 2006 22:22:50 GMT
View Forum Message <> Reply to Message

During driver initialization if the driver has an expensive
initialization routine usbatm starts a separate kernel thread for it.

In the driver cleanup routine the code waits to ensure the
initialization routine has finished.

Switching to the kthread api allowed some of the thread management
code to be removed.

In addition the kill_proc(SIGTERM, ...) in usbatm_usb_disconnect was
removed because it was absolutely pointless. The kernel thread did
not handle SIGTERM or any pending signals, so despite marking the
signal as pending it would never have been handled.

Signed-off-by: Eric W. Biederman <ebiederm@xmission.com>
---
drivers/usb/atm/usbatm.c | 24 ++++++----------
drivers/usb/atm/usbatm.h | 2 --
2 files changed, 6 insertions(+), 20 deletions(-)

```
diff --git a/drivers/usb/atm/usbatm.c b/drivers/usb/atm/usbatm.c
index ec63b0e..e6cd5e4 100644
--- a/drivers/usb/atm/usbatm.c
+++ b/drivers/usb/atm/usbatm.c
@@ -81,6 +81,7 @@
 #include <linux/stat.h>
 #include <linux/timer.h>
 #include <linux/wait.h>
+#include <linux/kthread.h>

 #ifdef VERBOSE_DEBUG
 static int usbatm_print_packet(const unsigned char *data, int len);
@@ -999,35 +1000,26 @@ static int usbatm_do_heavy_init(void *arg)
  struct usbatm_data *instance = arg;
  int ret;

- daemonize(instance->driver->driver_name);
- allow_signal(SIGTERM);
- instance->thread_pid = current->pid;
-
- complete(&instance->thread_started);
-
  ret = instance->driver->heavy_init(instance, instance->usb_intf);
```

```
  if (!ret)
   ret = usbatm_atm_init(instance);

- mutex_lock(&instance->serialize);
- instance->thread_pid = -1;
- mutex_unlock(&instance->serialize);

  complete_and_exit(&instance->thread_exited, ret);
 }

 static int usbatm_heavy_init(struct usbatm_data *instance)
 {
- int ret = kernel_thread(usbatm_do_heavy_init, instance, CLONE_KERNEL);
-
- if (ret < 0) {
+ struct task_struct *thread;
+ thread = kthread_run(usbatm_do_heavy_init, instance,
+     instance->driver->driver_name);
+ if (IS_ERR(thread)) {
+  int ret = PTR_ERR(thread);
   usb_err(instance, "%s: failed to create kernel_thread (%d)!\n", __func__, ret);
   return ret;
  }

- wait_for_completion(&instance->thread_started);
-
  return 0;
 }

@@ -1109,8 +1101,6 @@ int usbatm_usb_probe(struct usb_interface *intf, const struct
usb_device_id *id,
  kref_init(&instance->refcount);  /* dropped in usbatm_usb_disconnect */
  mutex_init(&instance->serialize);

- instance->thread_pid = -1;
- init_completion(&instance->thread_started);
  init_completion(&instance->thread_exited);

  INIT_LIST_HEAD(&instance->vcc_list);
@@ -1272,8 +1262,6 @@ void usbatm_usb_disconnect(struct usb_interface *intf)

  mutex_lock(&instance->serialize);
  instance->disconnected = 1;
- if (instance->thread_pid >= 0)
-  kill_proc(instance->thread_pid, SIGTERM, 1);
  mutex_unlock(&instance->serialize);

  wait_for_completion(&instance->thread_exited);
```

```
diff --git a/drivers/usb/atm/usbatm.h b/drivers/usb/atm/usbatm.h
index ff8551e..ab42355 100644
--- a/drivers/usb/atm/usbatm.h
+++ b/drivers/usb/atm/usbatm.h
@@ -176,8 +176,6 @@ struct usbatm_data {
  int disconnected;

  /* heavy init */
- int thread_pid;
- struct completion thread_started;
  struct completion thread_exited;

  /* ATM device */
--
1.4.4.1.g278f
```

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Cedric Le Goater on Wed, 13 Dec 2006 16:24:17 GMT
View Forum Message <> Reply to Message

Eric W. Biederman wrote:
> During driver initialization if the driver has an expensive
> initialization routine usbatm starts a separate kernel thread for it.
>
> In the driver cleanup routine the code waits to ensure the
> initialization routine has finished.
>
> Switching to the kthread api allowed some of the thread management
> code to be removed.
>
> In addition the kill_proc(SIGTERM, ...) in usbatm_usb_disconnect was
> removed because it was absolutely pointless. The kernel thread did
> not handle SIGTERM or any pending signals, so despite marking the
> signal as pending it would never have been handled.

are you sure that the heavy_init() routines don't handle pending
signals. they do firmware loading, etc. ?

> Signed-off-by: Eric W. Biederman <ebiederm@xmission.com>
> ---
> drivers/usb/atm/usbatm.c | 24 ++++++----------------
> drivers/usb/atm/usbatm.h | 2 --

```
>  2 files changed, 6 insertions(+), 20 deletions(-)
>
> diff --git a/drivers/usb/atm/usbatm.c b/drivers/usb/atm/usbatm.c
> index ec63b0e..e6cd5e4 100644
> --- a/drivers/usb/atm/usbatm.c
> +++ b/drivers/usb/atm/usbatm.c
> @@ -81,6 +81,7 @@
>  #include <linux/stat.h>
>  #include <linux/timer.h>
>  #include <linux/wait.h>
> +#include <linux/kthread.h>
>
>  #ifdef VERBOSE_DEBUG
>  static int usbatm_print_packet(const unsigned char *data, int len);
> @@ -999,35 +1000,26 @@ static int usbatm_do_heavy_init(void *arg)
>   struct usbatm_data *instance = arg;
>   int ret;
>
> - daemonize(instance->driver->driver_name);
> - allow_signal(SIGTERM);
> - instance->thread_pid = current->pid;
> -
> - complete(&instance->thread_started);
> -
>   ret = instance->driver->heavy_init(instance, instance->usb_intf);
>
>   if (!ret)
>    ret = usbatm_atm_init(instance);
>
> - mutex_lock(&instance->serialize);
> - instance->thread_pid = -1;
> - mutex_unlock(&instance->serialize);
>
>   complete_and_exit(&instance->thread_exited, ret);
>  }
>
>  static int usbatm_heavy_init(struct usbatm_data *instance)
>  {
> - int ret = kernel_thread(usbatm_do_heavy_init, instance, CLONE_KERNEL);
> -
> - if (ret < 0) {
> + struct task_struct *thread;
> + thread = kthread_run(usbatm_do_heavy_init, instance,
> +     instance->driver->driver_name);
> + if (IS_ERR(thread)) {
> +  int ret = PTR_ERR(thread);
>    usb_err(instance, "%s: failed to create kernel_thread (%d)!\n", __func__, ret);
>    return ret;
```

```
>   }
>
> - wait_for_completion(&instance->thread_started);
> -
>   return 0;
>  }
>
> @@ -1109,8 +1101,6 @@ int usbatm_usb_probe(struct usb_interface *intf, const struct
usb_device_id *id,
>   kref_init(&instance->refcount);  /* dropped in usbatm_usb_disconnect */
>   mutex_init(&instance->serialize);
>
> - instance->thread_pid = -1;
> - init_completion(&instance->thread_started);
>   init_completion(&instance->thread_exited);
>
>   INIT_LIST_HEAD(&instance->vcc_list);
> @@ -1272,8 +1262,6 @@ void usbatm_usb_disconnect(struct usb_interface *intf)
>
>   mutex_lock(&instance->serialize);
>   instance->disconnected = 1;
> - if (instance->thread_pid >= 0)
> -  kill_proc(instance->thread_pid, SIGTERM, 1);
>   mutex_unlock(&instance->serialize);
>
>   wait_for_completion(&instance->thread_exited);
> diff --git a/drivers/usb/atm/usbatm.h b/drivers/usb/atm/usbatm.h
> index ff8551e..ab42355 100644
> --- a/drivers/usb/atm/usbatm.h
> +++ b/drivers/usb/atm/usbatm.h
> @@ -176,8 +176,6 @@ struct usbatm_data {
>   int disconnected;
>
>   /* heavy init */
> - int thread_pid;
> - struct completion thread_started;
>   struct completion thread_exited;
>
>   /* ATM device */
```

Subject: Re: [PATCH] usbatm: Update to use the kthread api.

Posted by ebiederm on Wed, 13 Dec 2006 19:11:01 GMT
View Forum Message <> Reply to Message

Cedric Le Goater <clg@fr.ibm.com> writes:

> Eric W. Biederman wrote:
>> During driver initialization if the driver has an expensive
>> initialization routine usbatm starts a separate kernel thread for it.
>>
>> In the driver cleanup routine the code waits to ensure the
>> initialization routine has finished.
>>
>> Switching to the kthread api allowed some of the thread management
>> code to be removed.
>>
>> In addition the kill_proc(SIGTERM, ...) in usbatm_usb_disconnect was
>> removed because it was absolutely pointless. The kernel thread did
>> not handle SIGTERM or any pending signals, so despite marking the
>> signal as pending it would never have been handled.
>
> are you sure that the heavy_init() routines don't handle pending
> signals. they do firmware loading, etc. ?

Well I just took a quick look through them to be certain
and I don't see anything that would. Even inside of the guts of
request firmware. So I'm pretty certain that SIGTERM was something
originally copied from another kernel_thread implementation and
wound up being dead code.

Eric

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Duncan Sands on Thu, 14 Dec 2006 10:44:10 GMT
View Forum Message <> Reply to Message

Hi Eric, thanks for looking into this.

> During driver initialization if the driver has an expensive
> initialization routine usbatm starts a separate kernel thread for it.
>
> In the driver cleanup routine the code waits to ensure the
> initialization routine has finished.
>

> Switching to the kthread api allowed some of the thread management
> code to be removed.
>
> In addition the kill_proc(SIGTERM, ...) in usbatm_usb_disconnect was
> removed because it was absolutely pointless. The kernel thread did
> not handle SIGTERM or any pending signals, so despite marking the
> signal as pending it would never have been handled.

This is wrong, the signal is used. Let me explain the context, then why signals are important. USB ATM modem drivers register themselves with the usbatm core, which organizes the interaction between the USB layer, the ATM layer and the modem driver. Some modems require initialization that cannot be performed in the USB probe method. When I say "cannot" here, you need to understand that this is mainly about quality of service, though there are some correctness issues: initializing these modems takes typically 5 seconds or more. If the initialization was done in probe, all other USB device initialization/ disconnection would have to wait for it to finish (USB probe/disconnect is globally serialized, being run from the khubd kernel thread). This is unacceptable, so usbatm provides an easy way to have the extra initialization run from within it's own kernel thread: the modem driver registers a heavy_init method with usbatm; at the end of probe, heavy_init is run in its own kernel thread. In fact, I've been asked by Alan Stern to generalize this functionality into the USB core itself, since something like this is needed by a pile of USB drivers.

An important consideration: what if heavy_init is still running when the modem is disconnected? The disconnect method cannot exit until the kernel thread has exited; horrible mayhem could result otherwise. Thus disconnect has to wait for the kernel thread to finish. That means that the whole USB subsystem has to wait for the kernel thread to exit. This is problematic, from a quality of service point of view, if heavy_init takes a long time to finish. For example, the following line is executed by the heavy_init in speedtch.c:

```
msleep_interruptible(1000);
```

This is relatively mild, but already shows the problem: disconnect can take more than one second to exit. There are much worse cases (more on this later).

In short, the usbatm core needs a way to tell the heavy_init method that the game is up (due to disconnect). I chose to have it send a signal to the kernel thread. This seemed to be the simplest way. If the sending of a signal is removed, something else will have to replace it. Before I discuss how the signal is handled by existing heavy_init methods, I should point out that even if none of the

existing heavy_init methods made any use of the signal, it would still be wrong to remove it: a not-yet written heavy_init might well need to use it. But in fact the existing heavy_init routines do make use of the signal.

For example, consider speedtch_upload_firmware in speedtch.c. It does two things: it sends a bunch of urbs to the modem, and it performs the above msleep_interruptible. If disconnect is called, any urbs in progress promptly fail and any newly submitted urbs fail at once; thus the only thing that can take an appreciable amount of time is the msleep_interruptible. But this will also exit at once because of the signal sent by usbatm during disconnect. So, in this case, the signal reduces the maximum time the USB subsystem is blocked in disconnect from one second to zero seconds.

Now consider firmware loading, a nasty case. This is also done in heavy_init, and can take an infinite amount of time if the firmware is not found (the user can choose an infinite timeout); the default timeout of 10 seconds is already plenty long. Firmware loading also needs to exit at once if the modem is disconnected. You may well wonder how speedtch_heavy_init arranges to cancel firmware loading when the signal comes in. The answer is that it doesn't cancel it. But this is not a reason to remove the sending of the signal, it is a reason to improve speedtch_heavy_init. This is not so easy, because the firmware subsystem doesn't give the kernel any way of cancelling a firmware load once started, which is why it doesn't happen right now.

Once you accept that a signal needs to be sent, you can't remove all those completions etc that your patch deleted, because it introduces races: they are there to make sure that (a) signals are unblocked in the thread before the signal can possibly be sent, and (b) the signal is not sent to the wrong thread if the kernel thread exits at a badly chosen moment and the pid is recycled. I believe the current setup is race free, but please don't hesitate to correct me. If you want to get rid of the pid and instead use a pointer to the thread then avoiding race (b) becomes even more important, since then rather than shooting down the wrong thread you send a signal to a thread which no longer exists, surely a fatal mistake.

So it's a NACK for your current patch I'm afraid.

Best wishes,

Duncan Sands.

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Duncan Sands on Thu, 14 Dec 2006 10:45:56 GMT
View Forum Message <> Reply to Message

> Well I just took a quick look through them to be certain
> and I don't see anything that would. Even inside of the guts of
> request firmware. So I'm pretty certain that SIGTERM was something
> originally copied from another kernel_thread implementation and
> wound up being dead code.

Not at all, it was all written from scratch (so now you know who to
blame :)). And the signal *is* used, as explained in my reply to
your original email.

Ciao,

Duncan.

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Cedric Le Goater on Thu, 14 Dec 2006 11:13:34 GMT
View Forum Message <> Reply to Message

Hi Duncan,

Duncan Sands wrote:
>> Well I just took a quick look through them to be certain
>> and I don't see anything that would. Even inside of the guts of
>> request firmware. So I'm pretty certain that SIGTERM was something
>> originally copied from another kernel_thread implementation and
>> wound up being dead code.
>
> Not at all, it was all written from scratch (so now you know who to
> blame :)). And the signal *is* used, as explained in my reply to
> your original email.

Here's one I have been keeping for a while. Nothing really fancy :
basic replacement of kernel_thread and removal of the start
completion which is now covered in kthread.

Cheers,

C.

Subject: replace kernel_thread() with kthread_run() in usbatm

From: Cedric Le Goater <clg@fr.ibm.com>

This patch replaces the kernel_thread() with kthread_run() since
kernel_thread() is deprecated in drivers/modules.

Signed-off-by: Cedric Le Goater <clg@fr.ibm.com>
Cc: Duncan Sands <duncan.sands@free.fr>
Cc: linux-usb-users@lists.sourceforge.net
Cc: linux-usb-devel@lists.sourceforge.net

---
 drivers/usb/atm/usbatm.c | 26 ++++++++++++++----------
 drivers/usb/atm/usbatm.h | 3 +--
 2 files changed, 13 insertions(+), 16 deletions(-)

Index: 2.6.19-mm1/drivers/usb/atm/usbatm.c
===
--- 2.6.19-mm1.orig/drivers/usb/atm/usbatm.c
+++ 2.6.19-mm1/drivers/usb/atm/usbatm.c
@@ -81,6 +81,7 @@
 #include <linux/stat.h>
 #include <linux/timer.h>
 #include <linux/wait.h>
+#include <linux/kthread.h>

 #ifdef VERBOSE_DEBUG
 static int usbatm_print_packet(const unsigned char *data, int len);
@@ -999,11 +1000,7 @@ static int usbatm_do_heavy_init(void *ar
  struct usbatm_data *instance = arg;
  int ret;

- daemonize(instance->driver->driver_name);
  allow_signal(SIGTERM);
- instance->thread_pid = current->pid;
-
- complete(&instance->thread_started);

  ret = instance->driver->heavy_init(instance, instance->usb_intf);

@@ -1011,7 +1008,7 @@ static int usbatm_do_heavy_init(void *ar
   ret = usbatm_atm_init(instance);

  mutex_lock(&instance->serialize);
- instance->thread_pid = -1;
+ instance->thread_task = NULL;
  mutex_unlock(&instance->serialize);

```
  complete_and_exit(&instance->thread_exited, ret);
@@ -1019,15 +1016,17 @@ static int usbatm_do_heavy_init(void *ar

 static int usbatm_heavy_init(struct usbatm_data *instance)
 {
- int ret = kernel_thread(usbatm_do_heavy_init, instance, CLONE_KERNEL);
+ instance->thread_task = kthread_run(usbatm_do_heavy_init, instance,
+         instance->driver->driver_name);

- if (ret < 0) {
-  usb_err(instance, "%s: failed to create kernel_thread (%d)!\n", __func__, ret);
+ if (IS_ERR(instance->thread_task)) {
+  int ret = PTR_ERR(instance->thread_task);
+  usb_err(instance, "%s: failed to create kthread (%d)!\n",
+    __func__, ret);
+  instance->thread_task = NULL;
   return ret;
  }

- wait_for_completion(&instance->thread_started);
-
  return 0;
 }

@@ -1109,8 +1108,7 @@ int usbatm_usb_probe(struct usb_interfac
  kref_init(&instance->refcount);  /* dropped in usbatm_usb_disconnect */
  mutex_init(&instance->serialize);

- instance->thread_pid = -1;
- init_completion(&instance->thread_started);
+ instance->thread_task = NULL;
  init_completion(&instance->thread_exited);

  INIT_LIST_HEAD(&instance->vcc_list);
@@ -1272,8 +1270,8 @@ void usbatm_usb_disconnect(struct usb_in

  mutex_lock(&instance->serialize);
  instance->disconnected = 1;
- if (instance->thread_pid >= 0)
-  kill_proc(instance->thread_pid, SIGTERM, 1);
+ if (instance->thread_task)
+  send_sig(SIGTERM, instance->thread_task, 1);
  mutex_unlock(&instance->serialize);

  wait_for_completion(&instance->thread_exited);
Index: 2.6.19-mm1/drivers/usb/atm/usbatm.h
===================================================================
```

```
--- 2.6.19-mm1.orig/drivers/usb/atm/usbatm.h
+++ 2.6.19-mm1/drivers/usb/atm/usbatm.h
@@ -176,8 +176,7 @@ struct usbatm_data {
  int disconnected;

  /* heavy init */
- int thread_pid;
- struct completion thread_started;
+ struct task_struct *thread_task;
  struct completion thread_exited;

  /* ATM device */
```

_____

Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Duncan Sands on Thu, 14 Dec 2006 11:30:15 GMT
View Forum Message <> Reply to Message

Hi Cedric,

> Here's one I have been keeping for a while. Nothing really fancy :
> basic replacement of kernel_thread and removal of the start
> completion which is now covered in kthread.

the signal needs to be unblocked before the start completion,
since otherwise the signal might be sent while signals are
blocked (though this is extremely unlikely). So you can't
rely on kthread's start completion I'm afraid, because that
happens before the unblocking.

Ciao,

Duncan.

_____

Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by ebiederm on Thu, 14 Dec 2006 11:39:50 GMT
View Forum Message <> Reply to Message

Duncan Sands <baldrick@free.fr> writes:

> Hi Eric, thanks for looking into this.
>
>> During driver initialization if the driver has an expensive
>> initialization routine usbatm starts a separate kernel thread for it.
>>
>> In the driver cleanup routine the code waits to ensure the
>> initialization routine has finished.
>>
>> Switching to the kthread api allowed some of the thread management
>> code to be removed.
>>
>> In addition the kill_proc(SIGTERM, ...) in usbatm_usb_disconnect was
>> removed because it was absolutely pointless. The kernel thread did
>> not handle SIGTERM or any pending signals, so despite marking the
>> signal as pending it would never have been handled.

So first thank you for the review.

> This is wrong, the signal is used. Let me explain the context, then
> why signals are important. USB ATM modem drivers register themselves
> with the usbatm core, which organizes the interaction between the USB
> layer, the ATM layer and the modem driver. Some modems require
> initialization that cannot be performed in the USB probe method.
> When I say "cannot" here, you need to understand that this is mainly
> about quality of service, though there are some correctness issues:
> initializing these modems takes typically 5 seconds or more. If the
> initialization was done in probe, all other USB device initialization/
> disconnection would have to wait for it to finish (USB probe/disconnect
> is globally serialized, being run from the khubd kernel thread). This
> is unacceptable, so usbatm provides an easy way to have the extra
> initialization run from within it's own kernel thread: the modem driver
> registers a heavy_init method with usbatm; at the end of probe, heavy_init
> is run in its own kernel thread. In fact, I've been asked by Alan Stern
> to generalize this functionality into the USB core itself, since something
> like this is needed by a pile of USB drivers.
>
> An important consideration: what if heavy_init is still running
> when the modem is disconnected? The disconnect method cannot exit
> until the kernel thread has exited; horrible mayhem could result
> otherwise. Thus disconnect has to wait for the kernel thread to
> finish. That means that the whole USB subsystem has to wait for
> the kernel thread to exit. This is problematic, from a quality
> of service point of view, if heavy_init takes a long time to
> finish. For example, the following line is executed by the
> heavy_init in speedtch.c:
>

> msleep_interruptible(1000);
>
> This is relatively mild, but already shows the problem: disconnect
> can take more than one second to exit. There are much worse cases
> (more on this later).
>
> In short, the usbatm core needs a way to tell the heavy_init method
> that the game is up (due to disconnect). I chose to have it send
> a signal to the kernel thread. This seemed to be the simplest way.
> If the sending of a signal is removed, something else will have to
> replace it. Before I discuss how the signal is handled by existing
> heavy_init methods, I should point out that even if none of the
> existing heavy_init methods made any use of the signal, it would
> still be wrong to remove it: a not-yet written heavy_init might well
> need to use it. But in fact the existing heavy_init routines do
> make use of the signal.
>
> For example, consider speedtch_upload_firmware in speedtch.c. It
> does two things: it sends a bunch of urbs to the modem, and it
> performs the above msleep_interruptible. If disconnect is called,
> any urbs in progress promptly fail and any newly submitted urbs fail
> at once; thus the only thing that can take an appreciable amount of
> time is the msleep_interruptible. But this will also exit at once
> because of the signal sent by usbatm during disconnect. So, in this
> case, the signal reduces the maximum time the USB subsystem is blocked
> in disconnect from one second to zero seconds.
>
> Now consider firmware loading, a nasty case. This is also done in
> heavy_init, and can take an infinite amount of time if the firmware is
> not found (the user can choose an infinite timeout); the default timeout
> of 10 seconds is already plenty long. Firmware loading also needs to exit
> at once if the modem is disconnected. You may well wonder how
> speedtch_heavy_init arranges to cancel firmware loading when the signal
> comes in. The answer is that it doesn't cancel it. But this is not a
> reason to remove the sending of the signal, it is a reason to improve
> speedtch_heavy_init. This is not so easy, because the firmware subsystem
> doesn't give the kernel any way of cancelling a firmware load once started,
> which is why it doesn't happen right now.
>
> Once you accept that a signal needs to be sent, you can't remove all
> those completions etc that your patch deleted, because it introduces
> races: they are there to make sure that (a) signals are unblocked in the
> thread before the signal can possibly be sent, and (b) the signal is not
> sent to the wrong thread if the kernel thread exits at a badly chosen
> moment and the pid is recycled. I believe the current setup is race
> free, but please don't hesitate to correct me. If you want to get rid
> of the pid and instead use a pointer to the thread then avoiding race (b)
> becomes even more important, since then rather than shooting down the wrong

> thread you send a signal to a thread which no longer exists, surely a fatal
> mistake.

Actually I don't accept that a signal needs to be sent. I do accept
that the message needs to be delivered to stop things early.

The paradigm in a kthread world for waking up kernel threads is by
calling kthread_stop, and then for testing if a kernel thread should
stop is by calling kthread_should_stop.

Especially if you are looking at generalizing this code over all of
usb it should probably be using the current kernel best practices.

There is still an issue with msleep here that I completely concede.
In particular neither msleep nor msleep interruptible will actually be
awoken by kthread_stop. So it looks like we need a msleep_kthread
that will won't go back to sleep if after kthread_stop wakes it up.
Still unless I am blind that looks like a very minor change from where
we are now.

I think the reduction in complexity and the increase in uniformity
is most likely worth it.

If all else fails I'm happy with something simpler like Cedric's
patch which takes care of the things that I currently have a problem
with, but I'm willing to work through this to make it a through
cleanup.

Eric

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Duncan Sands on Thu, 14 Dec 2006 13:14:38 GMT
View Forum Message <> Reply to Message

Hi Eric,

> > ...
> > Once you accept that a signal needs to be sent, you can't remove all
> > those completions etc that your patch deleted, because it introduces
> > races: they are there to make sure that (a) signals are unblocked in the
> > thread before the signal can possibly be sent, and (b) the signal is not
> > sent to the wrong thread if the kernel thread exits at a badly chosen
> > moment and the pid is recycled. I believe the current setup is race

> > free, but please don't hesitate to correct me. If you want to get rid
> > of the pid and instead use a pointer to the thread then avoiding race (b)
> > becomes even more important, since then rather than shooting down the wrong
> > thread you send a signal to a thread which no longer exists, surely a fatal
> > mistake.
>
> Actually I don't accept that a signal needs to be sent. I do accept
> that the message needs to be delivered to stop things early.

I'm not in love with signals either, however...

> The paradigm in a kthread world for waking up kernel threads is by
> calling kthread_stop, and then for testing if a kernel thread should
> stop is by calling kthread_should_stop.

I considered this, but rejected it because of this comment:

 * kthread_stop - stop a thread created by kthread_create().
 * ... Your threadfn() must not call do_exit()
 * itself if you use this function! ...

and this one:

 * ... @threadfn can either call do_exit() directly if it is a
 * standalone thread for which noone will call kthread_stop(), or
 * return when 'kthread_should_stop()' is true (which means
 * kthread_stop() has been called).

Most of the time the kernel thread starts, performs heavy_init,
and exits. The above comments seem to imply that it is wrong
to call do_exit if kthread_stop might be called, and wrong to
return if kthread_stop has not been called. This seems to exclude
the case where kthread_stop is sometimes, but not always, called,
and the thread sometimes exits without kthread_stop having been
called. But perhaps I misunderstood, since it seems there is kthread
code to handle the case of a threadfn that returns without kthread_stop
having been called, witness this comment:
        /* It might have exited on its own, w/o kthread_stop. Check. */
It's still not clear to me, so if you can enlighten me, please do!

> Especially if you are looking at generalizing this code over all of
> usb it should probably be using the current kernel best practices.
>
> There is still an issue with msleep here that I completely concede.
> In particular neither msleep nor msleep interruptible will actually be
> awoken by kthread_stop. So it looks like we need a msleep_kthread
> that will won't go back to sleep if after kthread_stop wakes it up.
> Still unless I am blind that looks like a very minor change from where

> we are now.

Sure.

> I think the reduction in complexity and the increase in uniformity
> is most likely worth it.
>
> If all else fails I'm happy with something simpler like Cedric's
> patch which takes care of the things that I currently have a problem
> with, but I'm willing to work through this to make it a through
> cleanup.

You have a problem with the pid, right? Well, that is easily
cured in itself. I'll spin a patch for it a bit later, unless
someone else gets there first. And if you can confirm that kthread_stop
can be used in this situation (i.e. thread can spontaneously return
without kthread_stop) then I'm happy to convert everyone over to checking
kthread_should_stop.

Ciao,

Duncan.

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Cedric Le Goater on Thu, 14 Dec 2006 13:36:46 GMT
View Forum Message <> Reply to Message

I've added Christoph to Cc: for his expertise in kthread conversions.

> ...
>
> You have a problem with the pid, right? Well, that is easily
> cured in itself. I'll spin a patch for it a bit later, unless
> someone else gets there first. And if you can confirm that kthread_stop
> can be used in this situation (i.e. thread can spontaneously return
> without kthread_stop) then I'm happy to convert everyone over to checking
> kthread_should_stop.

C.

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Alan Stern on Thu, 14 Dec 2006 16:05:16 GMT
View Forum Message <> Reply to Message

On Thu, 14 Dec 2006, Eric W. Biederman wrote:

> Actually I don't accept that a signal needs to be sent. I do accept
> that the message needs to be delivered to stop things early.
>
> The paradigm in a kthread world for waking up kernel threads is by
> calling kthread_stop, and then for testing if a kernel thread should
> stop is by calling kthread_should_stop.
>
> Especially if you are looking at generalizing this code over all of
> usb it should probably be using the current kernel best practices.
>
> There is still an issue with msleep here that I completely concede.
> In particular neither msleep nor msleep interruptible will actually be
> awoken by kthread_stop. So it looks like we need a msleep_kthread
> that will won't go back to sleep if after kthread_stop wakes it up.
> Still unless I am blind that looks like a very minor change from where
> we are now.

Something else to think about. I've got a driver that starts up a kernel thread which calls vfs_read() and vfs_write() and relies on signals to interrupt the I/O operations when necessary. Perhaps this approach is fundamentally wrong, but I'm not sure how else to do it.

Alan Stern

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by ebiederm on Thu, 14 Dec 2006 22:51:42 GMT
View Forum Message <> Reply to Message

Grr. I missed this message first time through, the copy addressed to me did not make only the mailing list copy made it :(

Duncan Sands <baldrick@free.fr> writes:

> I'm not in love with signals either, however...

>
>> The paradigm in a kthread world for waking up kernel threads is by
>> calling kthread_stop, and then for testing if a kernel thread should
>> stop is by calling kthread_should_stop.
>
> I considered this, but rejected it because of this comment:
>
> * kthread_stop - stop a thread created by kthread_create().
> * ... Your threadfn() must not call do_exit()
> * itself if you use this function! ...
>
> and this one:
>
> * ... @threadfn can either call do_exit() directly if it is a
> * standalone thread for which noone will call kthread_stop(), or
> * return when 'kthread_should_stop()' is true (which means
> * kthread_stop() has been called).
>
> Most of the time the kernel thread starts, performs heavy_init,
> and exits. The above comments seem to imply that it is wrong
> to call do_exit if kthread_stop might be called, and wrong to
> return if kthread_stop has not been called. This seems to exclude
> the case where kthread_stop is sometimes, but not always, called,
> and the thread sometimes exits without kthread_stop having been
> called. But perhaps I misunderstood, since it seems there is kthread
> code to handle the case of a threadfn that returns without kthread_stop
> having been called, witness this comment:
>   /* It might have exited on its own, w/o kthread_stop. Check. */
> It's still not clear to me, so if you can enlighten me, please do!

This is a good point. I was going to say we could work around
this with checks in usbatm_do_heavy_init but that appears racy.

I guess I need to think a little more. I remember seeing this
and not worry about it because SIGTERM didn't seem to
be caught, so it didn't appear needed.

>> I think the reduction in complexity and the increase in uniformity
>> is most likely worth it.
>>
>> If all else fails I'm happy with something simpler like Cedric's
>> patch which takes care of the things that I currently have a problem
>> with, but I'm willing to work through this to make it a through
>> cleanup.
>
> You have a problem with the pid, right? Well, that is easily
> cured in itself. I'll spin a patch for it a bit later, unless
> someone else gets there first. And if you can confirm that kthread_stop

> can be used in this situation (i.e. thread can spontaneously return
> without kthread_stop) then I'm happy to convert everyone over to checking
> kthread_should_stop.

To be clear I have a problem with using numeric pids of kernel threads,
and with spawning threads from a possibly user space environment.
So on my hitlist are (kill_proc, daemonize, and kernel_thread).

If it the original process is a user space thread it is possible to
capture pieces of the user space environment unintentionally daemonize
is supposed to fix that but only does for the pieces of user
space environment that people have anticipated you can capture.

Eric

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Christoph Hellwig on Fri, 15 Dec 2006 09:16:55 GMT
View Forum Message <> Reply to Message

Can you add a little bit of context what all this is about, please?

On Thu, Dec 14, 2006 at 02:36:46PM +0100, Cedric Le Goater wrote:
>
> I've added Christoph to Cc: for his expertise in kthread conversions.
>
> > ...
> >
> > You have a problem with the pid, right? Well, that is easily
> > cured in itself. I'll spin a patch for it a bit later, unless
> > someone else gets there first. And if you can confirm that kthread_stop
> > can be used in this situation (i.e. thread can spontaneously return
> > without kthread_stop) then I'm happy to convert everyone over to checking
> > kthread_should_stop.

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.

Hi Eric,

> >> The paradigm in a kthread world for waking up kernel threads is by
> >> calling kthread_stop, and then for testing if a kernel thread should
> >> stop is by calling kthread_should_stop.
> >
> > I considered this, but rejected it because of this comment:
> >
> > * kthread_stop - stop a thread created by kthread_create().
> > * ... Your threadfn() must not call do_exit()
> > * itself if you use this function! ...
> >
> > and this one:
> >
> > * ... @threadfn can either call do_exit() directly if it is a
> > * standalone thread for which noone will call kthread_stop(), or
> > * return when 'kthread_should_stop()' is true (which means
> > * kthread_stop() has been called).
> >
> > Most of the time the kernel thread starts, performs heavy_init,
> > and exits. The above comments seem to imply that it is wrong
> > to call do_exit if kthread_stop might be called, and wrong to
> > return if kthread_stop has not been called. This seems to exclude
> > the case where kthread_stop is sometimes, but not always, called,
> > and the thread sometimes exits without kthread_stop having been
> > called. But perhaps I misunderstood, since it seems there is kthread
> > code to handle the case of a threadfn that returns without kthread_stop
> > having been called, witness this comment:
> >   /* It might have exited on its own, w/o kthread_stop. Check. */
> > It's still not clear to me, so if you can enlighten me, please do!
>
> This is a good point. I was going to say we could work around
> this with checks in usbatm_do_heavy_init but that appears racy.

presumably the problem is that if the thread has spontaneously exited, and
afterwards disconnect calls kthread_stop, then things go boom. The same
problem exists (though with lesser consequences) when sending a signal.
There is already code in usbatm to avoid this problem with signals. Why
not just recycle it in the kthread_stop case? I guess there is no
problem if you can guarantee that the following occurs:
if kthread_stop is ever called for the kthread, then the kthread only
exits after seeing kthread_should_stop return true.

> I guess I need to think a little more. I remember seeing this
> and not worry about it because SIGTERM didn't seem to
> be caught, so it didn't appear needed.

>
> >> I think the reduction in complexity and the increase in uniformity
> >> is most likely worth it.
> >>
> >> If all else fails I'm happy with something simpler like Cedric's
> >> patch which takes care of the things that I currently have a problem
> >> with, but I'm willing to work through this to make it a through
> >> cleanup.
> >
> > You have a problem with the pid, right? Well, that is easily
> > cured in itself. I'll spin a patch for it a bit later, unless
> > someone else gets there first. And if you can confirm that kthread_stop
> > can be used in this situation (i.e. thread can spontaneously return
> > without kthread_stop) then I'm happy to convert everyone over to checking
> > kthread_should_stop.
>
> To be clear I have a problem with using numeric pids of kernel threads,

Yes, this is a problem with usbatm at the moment.

> and with spawning threads from a possibly user space environment.

Not the case with usbatm. It is always spawned from khubd.

> So on my hitlist are (kill_proc, daemonize, and kernel_thread).
>
> If it the original process is a user space thread it is possible to
> capture pieces of the user space environment unintentionally daemonize
> is supposed to fix that but only does for the pieces of user
> space environment that people have anticipated you can capture.

Best wishes,

Duncan.

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by ebiederm on Fri, 15 Dec 2006 10:17:57 GMT
View Forum Message <> Reply to Message

Christoph Hellwig <hch@infradead.org> writes:

> Can you add a little bit of context what all this is about, please?
>

> On Thu, Dec 14, 2006 at 02:36:46PM +0100, Cedric Le Goater wrote:
>>
>> I've added Christoph to Cc: for his expertise in kthread conversions.
>>
>> > ...
>> >
>> > You have a problem with the pid, right? Well, that is easily
>> > cured in itself. I'll spin a patch for it a bit later, unless
>> > someone else gets there first. And if you can confirm that kthread_stop
>> > can be used in this situation (i.e. thread can spontaneously return
>> > without kthread_stop) then I'm happy to convert everyone over to checking
>> > kthread_should_stop.

In the long slow process to build container support in the linux kernel
one of the items on our todo list is the kernel_thread to kthread conversion.

While converting the usbatm driver we hit what is at least a partial snag.
I was hoping to remove the sending of signals along with the rest of
the conversion, but I hit a surprising use.

The usb atm drivers have some long running initializers (several seconds
potentially. So the infrastructure forks off a kernel thread to run them.

The code really does not care if the thread completes or does anything
else until a usb disconnect comes in. The in wants to wait suggest the
initialization code stop early and abort and then wait until the
initialization is done.

The practical problem is what is the best way to handle that case.

Can we use the kthread_should_stop() test in a thread that can
exit on it's own before kthread_stop is called?

Are signals the best available mechanism to request that a thread
stop that can exit on it's own.

If we don't suggest to the thread to stop having it call
complete_and_exit seems to the simplest race free solution. The
request to stop though makes things trickier.

Eric

_____

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Christoph Hellwig on Fri, 15 Dec 2006 10:35:01 GMT
View Forum Message <> Reply to Message

On Fri, Dec 15, 2006 at 03:17:57AM -0700, Eric W. Biederman wrote:
> While converting the usbatm driver we hit what is at least a partial snag.
> I was hoping to remove the sending of signals along with the rest of
> the conversion, but I hit a surprising use.
>
> The usb atm drivers have some long running initializers (several seconds
> potentially. So the infrastructure forks off a kernel thread to run them.
>
> The code really does not care if the thread completes or does anything
> else until a usb disconnect comes in. The in wants to wait suggest the
> initialization code stop early and abort and then wait until the
> initialization is done.
>
> The practical problem is what is the best way to handle that case.
>
> Can we use the kthread_should_stop() test in a thread that can
> exit on it's own before kthread_stop is called?

Right now it can't.

> Are signals the best available mechanism to request that a thread
> stop that can exit on it's own.

Defintly not. signals should be avoided in kernel threads at all
cost.

> If we don't suggest to the thread to stop having it call
> complete_and_exit seems to the simplest race free solution. The
> request to stop though makes things trickier.

I think the right fix is to encehance the kthread infrastructure to
gracefully handle the case where the threat has stopped by itself
and doesn't exist anymore at the time where we call kthread_stop.

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by ebiederm on Fri, 15 Dec 2006 10:45:40 GMT
View Forum Message <> Reply to Message

Christoph Hellwig <hch@infradead.org> writes:

> On Fri, Dec 15, 2006 at 03:17:57AM -0700, Eric W. Biederman wrote:
>
> I think the right fix is to encehance the kthread infrastructure to
> gracefully handle the case where the threat has stopped by itself
> and doesn't exist anymore at the time where we call kthread_stop.

Yep that is about where I thought we were at. Now we need to figure out
how to do that cleanly, and sanely.

Eric

_____

Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by ebiederm on Fri, 15 Dec 2006 10:54:00 GMT
View Forum Message <> Reply to Message

Duncan Sands <baldrick@free.fr> writes:

> Hi Eric,
>
> presumably the problem is that if the thread has spontaneously exited, and
> afterwards disconnect calls kthread_stop, then things go boom. The same
> problem exists (though with lesser consequences) when sending a signal.
> There is already code in usbatm to avoid this problem with signals. Why
> not just recycle it in the kthread_stop case? I guess there is no
> problem if you can guarantee that the following occurs:
> if kthread_stop is ever called for the kthread, then the kthread only
> exits after seeing kthread_should_stop return true.

I suspect we can recycle the locking on the signal sending code. At
least as a first pass. I have almost digested the problem sufficiently
to write some code. Maybe this weekend.

>> To be clear I have a problem with using numeric pids of kernel threads,
>
> Yes, this is a problem with usbatm at the moment.
>
>> and with spawning threads from a possibly user space environment.
>
> Not the case with usbatm. It is always spawned from khubd.

That is where I thought we were at, doing the conversion so it
is obvious and we can remove the use of kernel_thread and daemonize
would certainly be good. The more shared infrastructure we can
reasonably have the more likely the code will function correctly.

Eric

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Alan Stern on Fri, 15 Dec 2006 15:14:06 GMT
View Forum Message <> Reply to Message

On Fri, 15 Dec 2006, Christoph Hellwig wrote:

> > Are signals the best available mechanism to request that a thread
> > stop that can exit on it's own.
>
> Defintly not. signals should be avoided in kernel threads at all
> cost.

I have a driver that spawns a kernel thread (using kthread_create) which
does I/O by calling vfs_write and vfs_read. It relies on signals to
interrupt the I/O activity when necessary. Maybe this isn't a good way of
doing things, but I couldn't think of anything better.

Do you have any suggestions?

Alan Stern

P.S.: What is the reason for saying "signals should be avoided in kernel
threads at all cost"?

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Christoph Hellwig on Tue, 02 Jan 2007 11:11:19 GMT
View Forum Message <> Reply to Message

On Fri, Dec 15, 2006 at 10:14:06AM -0500, Alan Stern wrote:

> On Fri, 15 Dec 2006, Christoph Hellwig wrote:
>
> > > Are signals the best available mechanism to request that a thread
> > > stop that can exit on it's own.
> >
> > Defintly not. signals should be avoided in kernel threads at all
> > cost.
>
> I have a driver that spawns a kernel thread (using kthread_create) which
> does I/O by calling vfs_write and vfs_read. It relies on signals to
> interrupt the I/O activity when necessary. Maybe this isn't a good way of
> doing things, but I couldn't think of anything better.

Given that we have no other way to interrupt I/O then signals at those
lower level I don't see a way around the singals if you stick to that
higher level design.

> P.S.: What is the reason for saying "signals should be avoided in kernel
> threads at all cost"?

The probem with signals is that they can come from various sources, most
notably from random kill commands issues from userland. This defeats
the notion of a fixed thread lifetime under control of the owning module.
Of course this issue doesn't exist for you above useage where you'd
hopefully avoid allowing signals that could terminate the thread.

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Alan Stern on Tue, 02 Jan 2007 15:34:16 GMT

View Forum Message <> Reply to Message

On Tue, 2 Jan 2007, Christoph Hellwig wrote:

> > I have a driver that spawns a kernel thread (using kthread_create) which
> > does I/O by calling vfs_write and vfs_read. It relies on signals to
> > interrupt the I/O activity when necessary. Maybe this isn't a good way of
> > doing things, but I couldn't think of anything better.
>
> Given that we have no other way to interrupt I/O then signals at those
> lower level I don't see a way around the singals if you stick to that
> higher level design.

Okay.

> > P.S.: What is the reason for saying "signals should be avoided in kernel
> > threads at all cost"?
>
> The probem with signals is that they can come from various sources, most
> notably from random kill commands issues from userland. This defeats
> the notion of a fixed thread lifetime under control of the owning module.
> Of course this issue doesn't exist for you above useage where you'd
> hopefully avoid allowing signals that could terminate the thread.

In my case the situation is exactly the reverse: I _want_ to allow signals
to terminate the thread (as well as allowing signals to interrupt I/O).

The reason is simple enough. At system shutdown, if the thread isn't
terminated then it would continue to own an open file, preventing that
file's filesystem from being unmounted cleanly. Since people should be
able to unmount their disks during shutdown without having to unload
drivers first, the simplest solution is to allow the thread to respond to
the TERM signal normally sent by the shutdown scripts.

Since the thread is owned by the kernel, random kill commands won't have
any bad effect. Only kill commands sent by the superuser can terminate
the thread.

Alan Stern

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by ebiederm on Wed, 03 Jan 2007 09:05:28 GMT
View Forum Message <> Reply to Message

Alan Stern <stern@rowland.harvard.edu> writes:

> On Tue, 2 Jan 2007, Christoph Hellwig wrote:
>
>> > I have a driver that spawns a kernel thread (using kthread_create) which
>> > does I/O by calling vfs_write and vfs_read. It relies on signals to
>> > interrupt the I/O activity when necessary. Maybe this isn't a good way of
>> > doing things, but I couldn't think of anything better.
>>
>> Given that we have no other way to interrupt I/O then signals at those
>> lower level I don't see a way around the singals if you stick to that
>> higher level design.
>

> Okay.
>
>> > P.S.: What is the reason for saying "signals should be avoided in kernel
>> > threads at all cost"?
>>
>> The probem with signals is that they can come from various sources, most
>> notably from random kill commands issues from userland. This defeats
>> the notion of a fixed thread lifetime under control of the owning module.
>> Of course this issue doesn't exist for you above useage where you'd
>> hopefully avoid allowing signals that could terminate the thread.
>
> In my case the situation is exactly the reverse: I _want_ to allow signals
> to terminate the thread (as well as allowing signals to interrupt I/O).
>
> The reason is simple enough. At system shutdown, if the thread isn't
> terminated then it would continue to own an open file, preventing that
> file's filesystem from being unmounted cleanly. Since people should be
> able to unmount their disks during shutdown without having to unload
> drivers first, the simplest solution is to allow the thread to respond to
> the TERM signal normally sent by the shutdown scripts.
>
> Since the thread is owned by the kernel, random kill commands won't have
> any bad effect. Only kill commands sent by the superuser can terminate
> the thread.
>

Why in the world would a thread hold a file open for an indeterminate duration?
That seems very wrong.

I can just about understand reading a firmware file or something like that
and close the file afterwards. But unless you are worrying about a very small
window I think we have a problem here.

Eric

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by ebiederm on Wed, 03 Jan 2007 09:08:12 GMT
View Forum Message <> Reply to Message

Christoph Hellwig <hch@infradead.org> writes:

> Given that we have no other way to interrupt I/O then signals at those
> lower level I don't see a way around the singals if you stick to that

> higher level design.

It isn't hard to either modify signal_pending or the place where the
signal pending checks are to terminate things.

>> P.S.: What is the reason for saying "signals should be avoided in kernel
>> threads at all cost"?
>
> The probem with signals is that they can come from various sources, most
> notably from random kill commands issues from userland. This defeats
> the notion of a fixed thread lifetime under control of the owning module.
> Of course this issue doesn't exist for you above useage where you'd
> hopefully avoid allowing signals that could terminate the thread.

Right unless you can get a state where user space is not allowed to send
signals but the kernel is. But still reusing the concept if it doesn't quite
fit sounds like a definition mess.

Eric

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Alan Stern on Wed, 03 Jan 2007 17:52:42 GMT
View Forum Message <> Reply to Message

On Wed, 3 Jan 2007, Eric W. Biederman wrote:

> >> The probem with signals is that they can come from various sources, most
> >> notably from random kill commands issues from userland. This defeats
> >> the notion of a fixed thread lifetime under control of the owning module.
> >> Of course this issue doesn't exist for you above useage where you'd
> >> hopefully avoid allowing signals that could terminate the thread.
> >
> > In my case the situation is exactly the reverse: I _want_ to allow signals
> > to terminate the thread (as well as allowing signals to interrupt I/O).

I should have been clearer here. Signals don't terminate the thread; they
merely notify it to clean up and terminate itself.

> > The reason is simple enough. At system shutdown, if the thread isn't
> > terminated then it would continue to own an open file, preventing that

> > file's filesystem from being unmounted cleanly. Since people should be
> > able to unmount their disks during shutdown without having to unload
> > drivers first, the simplest solution is to allow the thread to respond to
> > the TERM signal normally sent by the shutdown scripts.
> >
> > Since the thread is owned by the kernel, random kill commands won't have
> > any bad effect. Only kill commands sent by the superuser can terminate
> > the thread.
> >
>
> Why in the world would a thread hold a file open for an indeterminate duration?
> That seems very wrong.

The thread uses the file to provide backing storage. Kind of like the
loop driver, except that my driver uses a higher-level interface into the
VFS than the loop driver does, for greater simplicity. You wouldn't say
that what the loop driver does is wrong, would you?

> I can just about understand reading a firmware file or something like that
> and close the file afterwards. But unless you are worrying about a very small
> window I think we have a problem here.

I don't follow. Why do you think there's a problem?

Alan Stern

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Duncan Sands on Wed, 03 Jan 2007 19:12:34 GMT
View Forum Message <> Reply to Message

On Friday 15 December 2006 11:45, Eric W. Biederman wrote:
> Christoph Hellwig <hch@infradead.org> writes:
>
> > On Fri, Dec 15, 2006 at 03:17:57AM -0700, Eric W. Biederman wrote:
> >
> > I think the right fix is to encehance the kthread infrastructure to
> > gracefully handle the case where the threat has stopped by itself
> > and doesn't exist anymore at the time where we call kthread_stop.
>
> Yep that is about where I thought we were at. Now we need to figure out
> how to do that cleanly, and sanely.

There's a completely different solution, which is to use a workqueue
instead of a kthread, with users providing a cancellation method.
Recall that the functionality is provided by usbatm to drivers:
they use it to perform slow initialization that is too slow to be
done in the probe method. They register with usbatm, providing a
"heavy_init" method. They could also provide a "heavy_cancel" method.
(Any special data that heavy_cancel needs can be stored in the
existing driver private data structure; this structure is already
passed to heavy_init). In the case of the speedtch driver, it
could place a completion in its private data structure; heavy_cancel
would just complete the completion. Rather than doing interruptible
sleeps, it can use wait_for_completion_timeout.

The only thing that worries me about this solution is... that you
can't shoot down firmware loading from userspace anymore. For
example, if heavy_init is blocked loading firmware when the system
is halted, it presumably won't react to the kill signal. Perhaps
it is unimportant; and if not, I guess I can just re-enable signals
in heavy_init.

Ciao,

Duncan.

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by ebiederm on Thu, 19 Apr 2007 02:13:56 GMT
View Forum Message <> Reply to Message

Alan Stern <stern@rowland.harvard.edu> writes:

> On Tue, 2 Jan 2007, Christoph Hellwig wrote:
>
>> > I have a driver that spawns a kernel thread (using kthread_create) which
>> > does I/O by calling vfs_write and vfs_read. It relies on signals to
>> > interrupt the I/O activity when necessary. Maybe this isn't a good way of
>> > doing things, but I couldn't think of anything better.
>>
>> Given that we have no other way to interrupt I/O then signals at those
>> lower level I don't see a way around the singals if you stick to that
>> higher level design.
>
> Okay.
>

>> > P.S.: What is the reason for saying "signals should be avoided in kernel
>> > threads at all cost"?
>>
>> The probem with signals is that they can come from various sources, most
>> notably from random kill commands issues from userland. This defeats
>> the notion of a fixed thread lifetime under control of the owning module.
>> Of course this issue doesn't exist for you above useage where you'd
>> hopefully avoid allowing signals that could terminate the thread.
>
> In my case the situation is exactly the reverse: I _want_ to allow signals
> to terminate the thread (as well as allowing signals to interrupt I/O).
>
> The reason is simple enough. At system shutdown, if the thread isn't
> terminated then it would continue to own an open file, preventing that
> file's filesystem from being unmounted cleanly. Since people should be
> able to unmount their disks during shutdown without having to unload
> drivers first, the simplest solution is to allow the thread to respond to
> the TERM signal normally sent by the shutdown scripts.

You need a real user space interface. Historically user space is
required to call unmount and the like, you should have a proper
shutdown interface and script as well.

> Since the thread is owned by the kernel, random kill commands won't have
> any bad effect. Only kill commands sent by the superuser can terminate
> the thread.

I'm putting the final touches on a patchset to finish converting
all kernel threads to the kthread API. So I have had a chance
to digest the arguments.

Upgrading kthreads to allow them to handle the when a thread exits
on it's own, and to set signal_pending so they terminate interruptible
sleeps was easy.

Handling signals from user space in kernel threads is a maintenance
disaster. It makes the kernel thread part of the ABI and makes it
so you can never change the implementation if user space comes to
rely on using them. If we don't handle signals kernel threads
remain an implementation detail leaving us free to change the
implementation later.

A pid namespace trivially changes the implementation making all kernel
threads invisible. Which means you can't send a signal to them.

So since sending signals to kernel threads is weird in terms of
semantics, it prevents from changing implementation details, and
because no kernel thread in the kernel appears to actually require it

at this time I refuse to support the concept.

Eric

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH] usbatm: Update to use the kthread api.
Posted by Alan Stern on Thu, 19 Apr 2007 14:56:51 GMT
View Forum Message <> Reply to Message

On Wed, 18 Apr 2007, Eric W. Biederman wrote:

> > In my case the situation is exactly the reverse: I _want_ to allow signals
> > to terminate the thread (as well as allowing signals to interrupt I/O).
> >
> > The reason is simple enough. At system shutdown, if the thread isn't
> > terminated then it would continue to own an open file, preventing that
> > file's filesystem from being unmounted cleanly. Since people should be
> > able to unmount their disks during shutdown without having to unload
> > drivers first, the simplest solution is to allow the thread to respond to
> > the TERM signal normally sent by the shutdown scripts.
>
> You need a real user space interface. Historically user space is
> required to call unmount and the like, you should have a proper
> shutdown interface and script as well.

Well, "kill <pid>" _is_ a real userspace interface.

Suppose you have a normal user application that runs a background process.
It's not a system service or anything like that, so it isn't controlled by
the runtime scripts in /etc/rc.d/init.d or wherever. How do you manage to
unmount the filesystems it uses at shutdown time? Simple -- you kill the
background process. Or rather, you rely on the shutdown script to kill
it for you.

My driver should work in the same way. Transparently. The user shouldn't
need to do anything special to shut down when the driver is loaded.

> I'm putting the final touches on a patchset to finish converting
> all kernel threads to the kthread API. So I have had a chance
> to digest the arguments.
>
> Upgrading kthreads to allow them to handle the when a thread exits
> on it's own, and to set signal_pending so they terminate interruptible
> sleeps was easy.

>
> Handling signals from user space in kernel threads is a maintenance
> disaster. It makes the kernel thread part of the ABI and makes it
> so you can never change the implementation if user space comes to
> rely on using them. If we don't handle signals kernel threads
> remain an implementation detail leaving us free to change the
> implementation later.
>
> A pid namespace trivially changes the implementation making all kernel
> threads invisible. Which means you can't send a signal to them.

What happens if there are user processes running in a PID namespace
different from the one used by the shutdown script? They won't get killed
at shutdown time, so the script won't be able to unmount the filesystems
they use.

> So since sending signals to kernel threads is weird in terms of
> semantics, it prevents from changing implementation details, and
> because no kernel thread in the kernel appears to actually require it
> at this time I refuse to support the concept.

_My_ thread is in the kernel and it actually requires it. Unless you can
suggest a suitable alternative mechanism. For example, if you provided a
way for the thread to allow its PID always to show up in the PID namespace
used by the shutdown scripts, that would resolve the problem. Or if you
suggested a way for the thread to hold an open file reference that would
automatically be closed when the filesystem was unmounted, that would work
too.

The alternative I don't like at all is to tell people using the driver
that they need to add

 grep -q g_file_storage /proc/modules && rmmod g_file_storage

somewhere in their /etc/rc.d/init.d/halt script.

Alan Stern

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers