

[patch 04/12] syslets: core code

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2007-02/msg10248.html>

- From: Ingo Molnar <mingo@xxxxxxxx>
 - Date: Wed, 28 Feb 2007 22:41:49 +0100
-

From: Ingo Molnar <mingo@xxxxxxxx>

the core syslet / async system calls infrastructure code.

Is built only if CONFIG_ASYNC_SUPPORT is enabled.

Signed-off-by: Ingo Molnar <mingo@xxxxxxxx>

Signed-off-by: Arjan van de Ven <arjan@xxxxxxxxxxxxxxxxxx>

kernel/Makefile | 1

kernel/async.c | 989 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

2 files changed, 990 insertions(+)

Index: linux/kernel/Makefile

=====

--- linux.orig/kernel/Makefile

+++ linux/kernel/Makefile

@@ -10,6 +10,7 @@ obj-y = sched.o fork.o exec_domain.o

kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o mutex.o \

hrtimer.o rwsem.o latency.o nsproxy.o srcu.o

+obj-\$(CONFIG_ASYNC_SUPPORT) += async.o

obj-\$(CONFIG_STACKTRACE) += stacktrace.o

obj-y += time/

obj-\$(CONFIG_DEBUG_MUTEXES) += mutex-debug.o

Index: linux/kernel/async.c

=====

--- /dev/null

+++ linux/kernel/async.c

@@ -0,0 +1,989 @@

+/*

+ * kernel/async.c

+ *

+ * The syslet and threadlet subsystem – asynchronous syscall and

+ * user-space code execution support.

+ *

+ * Started by Ingo Molnar:

+ *

+ * Copyright (C) 2007 Red Hat, Inc., Ingo Molnar <mingo@xxxxxxxx>

[patch 04/12] syslets: core code

```
+ *
+ * This file is released under the GPLv2.
+ *
+ * This code implements asynchronous syscalls via 'syslets'.
+ *
+ * Syslets consist of a set of 'syslet atoms' which are residing
+ * purely in user-space memory and have no kernel-space resource
+ * attached to them. These atoms can be linked to each other via
+ * pointers. Besides the fundamental ability to execute system
+ * calls, syslet atoms can also implement branches, loops and
+ * arithmetics.
+ *
+ * Thus syslets can be used to build small autonomous programs that
+ * the kernel can execute purely from kernel-space, without having
+ * to return to any user-space context. Syslets can be run by any
+ * unprivileged user-space application – they are executed safely
+ * by the kernel.
+ *
+ * "Threadlets" are the user-space equivalent of syslets: small
+ * functions of execution that user-space attempts/expects to execute
+ * without scheduling. If the threadlet nevertheless blocks, the kernel
+ * creates a real thread from it, and that thread is put aside sleeping.
+ * The 'head' context (the context that never blocks) returns to the
+ * original function that called the threadlet. Once the sleeping thread
+ * wakes up again (after it got for whatever it was waiting – IO, timeout,
+ * etc.) the function continues executing asynchronously, as a thread.
+ * A user-space completion ring connects these asynchronous function calls
+ * back to the head context.
+ */
+#include <linux/syscalls.h>
+#include <linux/syslet.h>
+#include <linux/delay.h>
+#include <linux/async.h>
+#include <linux/sched.h>
+#include <linux/init.h>
+#include <linux/err.h>
+
+#include <asm/uaccess.h>
+#include <asm/unistd.h>
+
+/*
+ * An async 'cachemiss context' is either busy, or it is ready.
+ * If it is ready, the 'head' might switch its user-space context
+ * to that ready thread anytime – so that if the ex-head blocks,
+ * one ready thread can become the next head and can continue to
+ * execute user-space code.
+ */
+static void
+__mark_async_thread_ready(struct async_thread *at, struct async_head *ah)
+{
+ list_del(&at->entry);
```

[patch 04/12] syslets: core code

```
+ list_add_tail(&at->entry, &ah->ready_async_threads);
+ if (list_empty(&ah->busy_async_threads))
+ wake_up(&ah->wait);
+}
+
+static void
+mark_async_thread_ready(struct async_thread *at, struct async_head *ah)
+{
+ spin_lock(&ah->lock);
+ __mark_async_thread_ready(at, ah);
+ spin_unlock(&ah->lock);
+}
+
+static void
+__mark_async_thread_busy(struct async_thread *at, struct async_head *ah)
+{
+ list_del(&at->entry);
+ list_add_tail(&at->entry, &ah->busy_async_threads);
+}
+
+static void
+mark_async_thread_busy(struct async_thread *at, struct async_head *ah)
+{
+ spin_lock(&ah->lock);
+ __mark_async_thread_busy(at, ah);
+ spin_unlock(&ah->lock);
+}
+
+static void
+__async_thread_init(struct task_struct *t, struct async_thread *at,
+ struct async_head *ah)
+{
+ INIT_LIST_HEAD(&at->entry);
+ at->exit = 0;
+ at->task = t;
+ at->ah = ah;
+
+ t->at = at;
+}
+
+static void
+async_thread_init(struct task_struct *t, struct async_thread *at,
+ struct async_head *ah)
+{
+ spin_lock(&ah->lock);
+ __async_thread_init(t, at, ah);
+ __mark_async_thread_ready(at, ah);
+ spin_unlock(&ah->lock);
+}
+
+static void
```

[patch 04/12] syslets: core code

```
+async_thread_exit(struct async_thread *at, struct task_struct *t)
+{
+ struct async_head *ah = at->ah;
+
+ spin_lock(&ah->lock);
+ list_del_init(&at->entry);
+ if (at->exit)
+ complete(&ah->exit_done);
+ t->at = NULL;
+ at->task = NULL;
+ spin_unlock(&ah->lock);
+}
+
+static struct async_thread *
+pick_ready_cachemiss_thread(struct async_head *ah)
+{
+ struct list_head *head = &ah->ready_async_threads;
+
+ if (list_empty(head))
+ return NULL;
+
+ return list_entry(head->next, struct async_thread, entry);
+}
+
+void __async_schedule(struct task_struct *t)
+{
+ struct async_thread *new_async_thread;
+ struct async_thread *async_ready;
+ struct async_head *ah = t->ah;
+ struct task_struct *new_task;
+
+ WARN_ON(!ah);
+ spin_lock(&ah->lock);
+
+ new_async_thread = pick_ready_cachemiss_thread(ah);
+ if (!new_async_thread)
+ goto out_unlock;
+
+ async_ready = t->async_ready;
+ WARN_ON(!async_ready);
+ t->async_ready = NULL;
+
+ new_task = new_async_thread->task;
+
+ move_user_context(new_task, t);
+ if (ah->restore_stack) {
+ set_task_stack_reg(new_task, ah->restore_stack);
+ WARN_ON(!ah->restore_ip);
+ task_ip_reg(new_task) = ah->restore_ip;
+ /*
+ * The return code 0 is needed to tell the
```

```

+ * head user-context that the threadlet went async:
+ */
+ task_ret_reg(new_task) = 0;
+ }
+
+ new_task->at = NULL;
+ t->ah = NULL;
+ new_task->ah = ah;
+ ah->user_task = new_task;
+
+ wake_up_process(new_task);
+
+ __async_thread_init(t, async_ready, ah);
+ __mark_async_thread_busy(t->at, ah);
+
+ out_unlock:
+ spin_unlock(&ah->lock);
+ }
+
+static void async_schedule(struct task_struct *t)
+{
+ if (t->async_ready)
+ __async_schedule(t);
+ }
+
+static long __exec_atom(struct task_struct *t, struct syslet_atom *atom)
+{
+ struct async_thread *async_ready_save;
+ long ret;
+
+ /*
+ * If user-space expects the syscall to schedule then
+ * (try to) switch user-space to another thread straight
+ * away and execute the syscall asynchronously:
+ */
+ if (unlikely(atom->flags & SYSLET_ASYNC))
+ async_schedule(t);
+ /*
+ * Does user-space want synchronous execution for this atom?:
+ */
+ async_ready_save = t->async_ready;
+ if (unlikely(atom->flags & SYSLET_SYNC))
+ t->async_ready = NULL;
+
+ if (unlikely(atom->nr >= atom->nr_syscalls))
+ return -ENOSYS;
+
+ ret = atom->call_table[atom->nr](atom->args[0], atom->args[1],
+ atom->args[2], atom->args[3],
+ atom->args[4], atom->args[5]);
+
+ }

```

[patch 04/12] syslets: core code

```
+ if (atom->ret_ptr && put_user(ret, atom->ret_ptr))
+ return -EFAULT;
+
+ if (t->ah)
+ t->async_ready = async_ready_save;
+
+ return ret;
+}
+
+/*
+ * Arithmetics syscall, add a value to a user-space memory location.
+ *
+ * Generic C version - in case the architecture has not implemented it
+ * in assembly.
+ */
+asmlinkage __attribute__((weak)) long
+sys_umem_add(unsigned long __user *uptr, unsigned long inc)
+{
+ unsigned long val, new_val;
+
+ if (get_user(val, uptr))
+ return -EFAULT;
+ /*
+ * inc == 0 means 'read memory value':
+ */
+ if (!inc)
+ return val;
+
+ new_val = val + inc;
+ if (__put_user(new_val, uptr))
+ return -EFAULT;
+
+ return new_val;
+}
+
+/*
+ * Open-coded because this is a very hot codepath during syslet
+ * execution and every cycle counts ...
+ *
+ * [ NOTE: it's an explicit fastcall because optimized assembly code
+ * might depend on this. There are some kernels that disable regparm,
+ * so lets not break those if possible. ]
+ */
+fastcall __attribute__((weak)) long
+copy_uatom(struct syslet_atom *atom, struct syslet_uatom __user *uatom)
+{
+ unsigned long __user *arg_ptr;
+ long ret = 0;
+
+ if (!access_ok(VERIFY_READ, uatom, sizeof(*uatom)))
+ return -EFAULT;
```

[patch 04/12] syslets: core code

```
+
+ ret = __get_user(atom->nr, &uatom->nr);
+ ret |= __get_user(atom->ret_ptr, (long __user **)&uatom->ret_ptr);
+ ret |= __get_user(atom->flags, (unsigned long __user *)&uatom->flags);
+ ret |= __get_user(atom->next,
+ (struct syslet_uatom __user **)&uatom->next);
+
+ memset(atom->args, 0, sizeof(atom->args));
+
+ ret |= __get_user(arg_ptr, (unsigned long __user **)&uatom->arg_ptr[0]);
+ if (!arg_ptr)
+ return ret;
+ if (!access_ok(VERIFY_READ, arg_ptr, sizeof(*arg_ptr)))
+ return -EFAULT;
+ ret |= __get_user(atom->args[0], arg_ptr);
+
+ ret |= __get_user(arg_ptr, (unsigned long __user **)&uatom->arg_ptr[1]);
+ if (!arg_ptr)
+ return ret;
+ if (!access_ok(VERIFY_READ, arg_ptr, sizeof(*arg_ptr)))
+ return -EFAULT;
+ ret |= __get_user(atom->args[1], arg_ptr);
+
+ ret |= __get_user(arg_ptr, (unsigned long __user **)&uatom->arg_ptr[2]);
+ if (!arg_ptr)
+ return ret;
+ if (!access_ok(VERIFY_READ, arg_ptr, sizeof(*arg_ptr)))
+ return -EFAULT;
+ ret |= __get_user(atom->args[2], arg_ptr);
+
+ ret |= __get_user(arg_ptr, (unsigned long __user **)&uatom->arg_ptr[3]);
+ if (!arg_ptr)
+ return ret;
+ if (!access_ok(VERIFY_READ, arg_ptr, sizeof(*arg_ptr)))
+ return -EFAULT;
+ ret |= __get_user(atom->args[3], arg_ptr);
+
+ ret |= __get_user(arg_ptr, (unsigned long __user **)&uatom->arg_ptr[4]);
+ if (!arg_ptr)
+ return ret;
+ if (!access_ok(VERIFY_READ, arg_ptr, sizeof(*arg_ptr)))
+ return -EFAULT;
+ ret |= __get_user(atom->args[4], arg_ptr);
+
+ ret |= __get_user(arg_ptr, (unsigned long __user **)&uatom->arg_ptr[5]);
+ if (!arg_ptr)
+ return ret;
+ if (!access_ok(VERIFY_READ, arg_ptr, sizeof(*arg_ptr)))
+ return -EFAULT;
+ ret |= __get_user(atom->args[5], arg_ptr);
+
```

```

+ return ret;
+ }
+
+ /*
+ * Should the next atom run, depending on the return value of
+ * the current atom – or should we stop execution?
+ */
+ static int run_next_atom(struct syslet_atom *atom, long ret)
+ {
+ switch (atom->flags & SYSLET_STOP_MASK) {
+ case SYSLET_STOP_ON_NONZERO:
+ if (!ret)
+ return 1;
+ return 0;
+ case SYSLET_STOP_ON_ZERO:
+ if (ret)
+ return 1;
+ return 0;
+ case SYSLET_STOP_ON_NEGATIVE:
+ if (ret >= 0)
+ return 1;
+ return 0;
+ case SYSLET_STOP_ON_NON_POSITIVE:
+ if (ret > 0)
+ return 1;
+ return 0;
+ }
+ return 1;
+ }
+
+ static struct syslet_uatom __user *
+ next_uatom(struct syslet_atom *atom, struct syslet_uatom *uatom, long ret)
+ {
+ /*
+ * If the stop condition is false then continue
+ * to atom->next:
+ */
+ if (run_next_atom(atom, ret))
+ return atom->next;
+ /*
+ * Special-case: if the stop condition is true and the atom
+ * has SKIP_TO_NEXT_ON_STOP set, then instead of
+ * stopping we skip to the atom directly after this atom
+ * (in linear address-space).
+ *
+ * This, combined with the atom->next pointer and the
+ * stop condition flags is what allows true branches and
+ * loops in syslets:
+ */
+ if (atom->flags & SYSLET_SKIP_TO_NEXT_ON_STOP)
+ return uatom + 1;

```

```

+
+ return NULL;
+}
+
+/*
+ * If user-space requested a completion event then put the last
+ * executed uatom into the completion ring:
+ */
+static long
+completion_event(struct async_head *ah, struct task_struct *t,
+ void __user *event, struct async_head_user __user *ahu)
+{
+ unsigned long ring_size_bytes, max_ring_idx, kernel_ring_idx;
+ struct syslet_uatom __user *slot_val = NULL;
+ u64 __user *completion_ring, *ring_slot;
+
+ WARN_ON(!t->at);
+ WARN_ON(t->ah);
+
+ if (!access_ok(VERIFY_WRITE, ahu, sizeof(*ahu)))
+ return -EFAULT;
+
+ if (__get_user(completion_ring,
+ (u64 __user **)&ahu->completion_ring_ptr))
+ return -EFAULT;
+ if (__get_user(ring_size_bytes,
+ (unsigned long __user *)&ahu->ring_size_bytes))
+ return -EFAULT;
+ if (!ring_size_bytes)
+ return -EINVAL;
+
+ max_ring_idx = ring_size_bytes / sizeof(u64);
+ if (ring_size_bytes != max_ring_idx * sizeof(u64))
+ return -EINVAL;
+ /*
+ * We pre-check the ring pointer, so that in the fastpath
+ * we can use __get_user():
+ */
+ if (!access_ok(VERIFY_WRITE, completion_ring, ring_size_bytes))
+ return -EFAULT;
+
+ mutex_lock(&ah->completion_lock);
+ /*
+ * Asynchron threads can complete in parallel so use the
+ * head-lock to serialize:
+ */
+ if (__get_user(kernel_ring_idx,
+ (unsigned long __user *)&ahu->kernel_ring_idx))
+ goto fault_unlock;
+ if (kernel_ring_idx >= max_ring_idx)
+ goto err_unlock;

```

```

+
+ ring_slot = completion_ring + kernel_ring_idx;
+ if (__get_user(slot_val, (struct syslet_uatom __user **)ring_slot))
+ goto fault_unlock;
+ /*
+ * User-space submitted more work than what fits into the
+ * completion ring – do not stomp over it silently and signal
+ * the error condition:
+ */
+ if (slot_val)
+ goto err_unlock;
+
+ slot_val = event;
+ if (__put_user(slot_val, (struct syslet_uatom __user **)ring_slot))
+ goto fault_unlock;
+ /*
+ * Update the ring index:
+ */
+ kernel_ring_idx++;
+ if (kernel_ring_idx == max_ring_idx)
+ kernel_ring_idx = 0;
+
+ if (__put_user(kernel_ring_idx, &ah->kernel_ring_idx))
+ goto fault_unlock;
+
+ /*
+ * See whether the async-head is waiting and needs a wakeup:
+ */
+ if (ah->events_left) {
+ if (!--ah->events_left) {
+ /*
+ * We first unlock the mutex – to reduce the size
+ * of the critical section. We have a safe
+ * reference to 'ah':
+ */
+ mutex_unlock(&ah->completion_lock);
+ wake_up(&ah->wait);
+ goto out;
+ }
+ }
+
+ mutex_unlock(&ah->completion_lock);
+ out:
+ return 0;
+
+ fault_unlock:
+ mutex_unlock(&ah->completion_lock);
+
+ return -EFAULT;
+
+ err_unlock:

```

```

+ mutex_unlock(&ah->completion_lock);
+
+ return -EINVAL;
+}
+
+/*
+ * This is the main syslet atom execution loop. This fetches atoms
+ * and executes them until it runs out of atoms or until the
+ * exit condition becomes false:
+ */
+static struct syslet_uatom __user *
+exec_atom(struct async_head *ah, struct task_struct *t,
+ struct syslet_uatom __user *uatom,
+ struct async_head_user __user *ahu,
+ syscall_fn_t *call_table,
+ unsigned int nr_syscalls)
+{
+ struct syslet_uatom __user *last_uatom;
+ struct syslet_atom atom;
+ long ret;
+
+ atom.call_table = call_table;
+ atom.nr_syscalls = nr_syscalls;
+
+ run_next:
+ if (unlikely(copy_uatom(&atom, uatom)))
+ return ERR_PTR(-EFAULT);
+
+ last_uatom = uatom;
+ ret = __exec_atom(t, &atom);
+ if (unlikely(signal_pending(t)))
+ goto stop;
+ if (need_resched())
+ cond_resched();
+
+ uatom = next_uatom(&atom, uatom, ret);
+ if (uatom)
+ goto run_next;
+ stop:
+ /*
+ * We do completion only in async context:
+ */
+ if (t->at && !(atom.flags & SYSLET_NO_COMPLETE)) {
+ if (completion_event(ah, t, last_uatom, ahu))
+ return ERR_PTR(-EFAULT);
+ }
+
+ return last_uatom;
+}
+
+static long

```

[patch 04/12] syslets: core code

```
+cachemiss_loop(struct async_thread *at, struct async_head *ah,
+ struct task_struct *t)
+{
+ for (;;) {
+ mark_async_thread_busy(at, ah);
+ set_task_state(t, TASK_INTERRUPTIBLE);
+ if (unlikely(t->ah || at->exit || signal_pending(t)))
+ break;
+ mark_async_thread_ready(at, ah);
+ schedule();
+ }
+ t->state = TASK_RUNNING;
+
+ async_thread_exit(at, t);
+
+ if (at->exit)
+ do_exit(0);
+
+ if (!t->ah) {
+ /*
+ * Cachemiss threads return to one given
+ * user-space instruction address and stack
+ * pointer:
+ */
+ set_task_stack_reg(t, at->user_stack);
+ task_ip_reg(t) = at->user_ip;
+
+ return -1;
+ }
+ return 0;
+}
+
+/*
+ * This is what a newly created cachemiss thread executes for the
+ * first time: initialize, pick up the user stack/IP addresses from
+ * the head and then execute the cachemiss loop. If the cachemiss
+ * loop returns then we return back to user-space:
+ */
+static long cachemiss_thread(void *data)
+{
+ struct pt_regs *head_regs, *regs;
+ struct task_struct *t = current;
+ struct async_head *ah = data;
+ struct async_thread *at;
+ int ret;
+
+ at = &t->__at;
+ async_thread_init(t, at, ah);
+
+ /*
+ * Clone the head thread's user-space pregs over,
```

```

+ * now that we are in kernel-space:
+ */
+ head_regs = task_pt_regs(ah->user_task);
+ regs = task_pt_regs(t);
+
+ *regs = *head_regs;
+ ret = get_user(at->user_stack, ah->new_stackp);
+ WARN_ON(ret);
+ /*
+ * Clear the stack pointer, signalling to user-space that
+ * this thread stack has been used up:
+ */
+ ret = put_user(0, ah->new_stackp);
+ WARN_ON(ret);
+
+ complete(&ah->start_done);
+
+ return cachemiss_loop(at, ah, t);
+}
+
+/**
+ * sys_async_thread - do work as an async cachemiss thread again
+ *
+ * @event: completion event
+ * @ahu: async head
+ *
+ * If an async thread has returned back to user-space (due to say
+ * a signal) then it is a 'busy' thread during that period. It
+ * can again offer itself into the cachemiss pool by calling this
+ * syscall:
+ */
+asmlinkage long
+sys_async_thread(void __user *event, struct async_head_user __user *ahu)
+{
+ struct task_struct *t = current;
+ struct async_thread *at = t->at;
+ struct async_head *ah = t->__at.ah;
+
+ /*
+ * Only async threads are allowed to do this:
+ */
+ if (!ah || t->ah)
+ return -EINVAL;
+
+ /*
+ * A threadlet might want to signal a completion event:
+ */
+ if (event) {
+ /*
+ * threadlet - make sure the stack is never used
+ * again by this thread:

```

```

+ */
+ set_task_stack_reg(t, 0x11111111);
+ task_ip_reg(t) = 0x22222222;
+
+ if (completion_event(ah, t, event, ahu))
+ return -EFAULT;
+ }
+ /*
+ * If a cachemiss threadlet calls sys_async_thread()
+ * then we first have to mark it ready:
+ */
+ if (at) {
+ mark_async_thread_ready(at, ah);
+ } else {
+ at = &t->__at;
+ WARN_ON(!at->ah);
+
+ async_thread_init(t, at, ah);
+ }
+
+ return cachemiss_loop(at, at->ah, t);
+}
+
+/*
+ * Initialize the in-kernel async head, based on the user-space async
+ * head:
+ */
+static long
+async_head_init(struct task_struct *t, struct async_head_user __user *ahu)
+{
+ struct async_head *ah;
+
+ ah = &t->__ah;
+
+ spin_lock_init(&ah->lock);
+ INIT_LIST_HEAD(&ah->ready_async_threads);
+ INIT_LIST_HEAD(&ah->busy_async_threads);
+ init_waitqueue_head(&ah->wait);
+ mutex_init(&ah->completion_lock);
+ ah->events_left = 0;
+ ah->ahu = NULL;
+ ah->new_stackp = NULL;
+ ah->new_ip = 0;
+ ah->restore_stack = 0;
+ ah->restore_ip = 0;
+ ah->user_task = t;
+ t->ah = ah;
+
+ return 0;
+}
+

```

[patch 04/12] syslets: core code

```
+/*
+ * If the head cache-misses then it will become a cachemiss
+ * thread after having finished its current syslet. If it
+ * returns to user-space after that point (to handle a signal
+ * for example) then it will need a thread stack of its own:
+ */
+static long init_head(struct async_head *ah, struct task_struct *t,
+ struct async_head_user __user *ahu)
+{
+ unsigned long head_stack, head_ip;
+
+ if (get_user(head_stack, (unsigned long __user *)&ahu->head_stack))
+ return -EFAULT;
+ if (get_user(head_ip, (unsigned long __user *)&ahu->head_ip))
+ return -EFAULT;
+ t->__at.user_stack = head_stack;
+ t->__at.user_ip = head_ip;
+
+ return async_head_init(t, ahu);
+}
+
+/*
+ * Simple limit and pool management mechanism for now:
+ */
+static long
+refill_cachemiss_pool(struct async_head *ah, struct task_struct *t,
+ struct async_head_user __user *ahu)
+{
+ unsigned long new_ip;
+ int pid, ret;
+
+ init_completion(&ah->start_done);
+ ah->new_stackp = (unsigned long __user *)&ahu->new_thread_stack;
+ ret = get_user(new_ip, (unsigned long __user *)&ahu->new_thread_ip);
+ WARN_ON(ret);
+ ah->new_ip = new_ip;
+
+ pid = create_async_thread(cachemiss_thread, (void *)ah,
+ CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND |
+ CLONE_THREAD | CLONE_SYSVSEM);
+ if (pid < 0)
+ return pid;
+
+ wait_for_completion(&ah->start_done);
+ ah->new_stackp = NULL;
+ ah->new_ip = 0;
+
+ return 0;
+}
+
+/**
```

[patch 04/12] syslets: core code

```
+ * sys_async_exec – execute a syslet.
+ *
+ * returns the uatom that was last executed, if the kernel was able to
+ * execute the syslet synchronously, or NULL if the syslet became
+ * asynchronous. (in the latter case syslet completion will be notified
+ * via the completion ring)
+ *
+ * (Various errors might also be returned via the usual negative numbers.)
+ */
+static struct syslet_uatom __user *
+_sys_async_exec(struct syslet_uatom __user *uatom,
+ struct async_head_user __user *ahu,
+ syscall_fn_t *call_table,
+ unsigned int nr_syscalls)
+{
+ struct syslet_uatom __user *ret;
+ struct task_struct *t = current;
+ struct async_head *ah = t->ah;
+ struct async_thread *at = &t->__at;
+
+ /*
+ * Do not allow recursive calls of sys_async_exec():
+ */
+ if (async_syscall(t))
+ return ERR_PTR(-ENOSYS);
+
+ if (!uatom || !ahu || !ahu->new_thread_stack)
+ return ERR_PTR(-EINVAL);
+
+ if (unlikely(!ah)) {
+ ret = (void *)init_head(ah, t, ahu);
+ if (ret)
+ return ret;
+ ah = t->ah;
+ }
+
+ if (unlikely(list_empty(&ah->ready_async_threads))) {
+ ret = (void *)refill_cachemiss_pool(ah, t, ahu);
+ if (ret)
+ return ret;
+ }
+
+ t->async_ready = at;
+ ah->ahu = ahu;
+
+ ret = exec_atom(ah, t, uatom, ahu, call_table, nr_syscalls);
+
+ /*
+ * Are we still executing as head?
+ */
+ if (t->ah) {
```

```

+ t->async_ready = NULL;
+
+ return ret;
+ }
+
+ /*
+ * We got turned into a cachemiss thread,
+ * enter the cachemiss loop:
+ */
+ set_task_state(t, TASK_INTERRUPTIBLE);
+ mark_async_thread_ready(at, ah);
+
+ return ERR_PTR(cachemiss_loop(at, ah, t));
+}
+
+asmlinkage struct syslet_uatom __user *
+sys_async_exec(struct syslet_uatom __user *uatom,
+ struct async_head_user __user *ahu)
+{
+ return __sys_async_exec(uatom, ahu, sys_call_table, NR_syscalls);
+}
+
+#ifdef CONFIG_COMPAT
+
+asmlinkage struct syslet_uatom __user *
+compat_sys_async_exec(struct syslet_uatom __user *uatom,
+ struct async_head_user __user *ahu)
+{
+ return __sys_async_exec(uatom, ahu, compat_sys_call_table,
+ compat_NR_syscalls);
+}
+
+#endif
+
+/**
+ * sys_async_wait – wait for async completion events
+ *
+ * This syscall waits for @min_wait_events syslet completion events
+ * to finish or for all async processing to finish (whichever
+ * comes first).
+ */
+asmlinkage long
+sys_async_wait(unsigned long min_wait_events, unsigned long user_ring_idx,
+ struct async_head_user __user *ahu)
+{
+ struct task_struct *t = current;
+ struct async_head *ah = t->ah;
+ unsigned long kernel_ring_idx;
+
+ /*
+ * Do not allow async waiting:

```

```

+ */
+ if (async_syscall(t))
+ return -ENOSYS;
+ if (!ah)
+ return -EINVAL;
+
+ mutex_lock(&ah->completion_lock);
+ if (get_user(kernel_ring_idx,
+ (unsigned long __user *)&ahu->kernel_ring_idx))
+ goto err_unlock;
+ /*
+ * Account any completions that happened since user-space
+ * checked the ring:
+ */
+ ah->events_left = min_wait_events - (kernel_ring_idx - user_ring_idx);
+ mutex_unlock(&ah->completion_lock);
+
+ return wait_event_interruptible(ah->wait,
+ list_empty(&ah->busy_async_threads) || ah->events_left <= 0);
+
+ err_unlock:
+ mutex_unlock(&ah->completion_lock);
+ return -EFAULT;
+}
+
+asmlinkage long
+sys_threadlet_on(unsigned long restore_stack,
+ unsigned long restore_ip,
+ struct async_head_user __user *ahu)
+{
+ struct task_struct *t = current;
+ struct async_head *ah = t->ah;
+ struct async_thread *at = &t->__at;
+ long ret;
+
+ /*
+ * Do not allow recursive calls of sys_threadlet_on():
+ */
+ if (t->async_ready || t->at)
+ return -EINVAL;
+
+ if (unlikely(!ah)) {
+ ret = init_head(ah, t, ahu);
+ if (ret)
+ return ret;
+ ah = t->ah;
+ }
+
+ if (unlikely(list_empty(&ah->ready_async_threads))) {
+ ret = refill_cachemiss_pool(ah, t, ahu);
+ if (ret)

```

```

+ return ret;
+ }
+
+ t->async_ready = at;
+ ah->restore_stack = restore_stack;
+ ah->restore_ip = restore_ip;
+
+ ah->ahu = ahu;
+
+ return 0;
+}
+
+asmlinkage long sys_threadlet_off(void)
+{
+ struct task_struct *t = current;
+ struct async_head *ah = t->ah;
+
+ /*
+ * Are we still executing as head?
+ */
+ if (ah) {
+ t->async_ready = NULL;
+
+ return 1;
+ }
+
+ /*
+ * We got turned into a cachemiss thread,
+ * return to user-space, which can do
+ * the notification, etc:
+ */
+ return 0;
+}
+
+static void __notify_async_thread_exit(struct async_thread *at,
+ struct async_head *ah)
+{
+ list_del_init(&at->entry);
+ at->exit = 1;
+ init_completion(&ah->exit_done);
+ wake_up_process(at->task);
+}
+
+static void stop_cachemiss_threads(struct async_head *ah)
+{
+ struct async_thread *at;
+
+repeat:
+ spin_lock(&ah->lock);
+ list_for_each_entry(at, &ah->ready_async_threads, entry) {
+

```

```

+ __notify_async_thread_exit(at, ah);
+ spin_unlock(&ah->lock);
+
+ wait_for_completion(&ah->exit_done);
+
+ goto repeat;
+ }
+
+ list_for_each_entry(at, &ah->busy_async_threads, entry) {
+
+ __notify_async_thread_exit(at, ah);
+ spin_unlock(&ah->lock);
+
+ wait_for_completion(&ah->exit_done);
+
+ goto repeat;
+ }
+ spin_unlock(&ah->lock);
+}
+
+static void async_head_exit(struct async_head *ah, struct task_struct *t)
+{
+ stop_cachemiss_threads(ah);
+ WARN_ON(!list_empty(&ah->ready_async_threads));
+ WARN_ON(!list_empty(&ah->busy_async_threads));
+ WARN_ON(spin_is_locked(&ah->lock));
+
+ t->ah = NULL;
+}
+
+/*
+ * fork()-time initialization:
+ */
+void async_init(struct task_struct *t)
+{
+ t->at = NULL;
+ t->async_ready = NULL;
+ t->ah = NULL;
+ t->__at.ah = NULL;
+ t->__at.user_stack = 0;
+}
+
+/*
+ * do_exit()-time cleanup:
+ */
+void async_exit(struct task_struct *t)
+{
+ struct async_thread *at = t->at;
+ struct async_head *ah = t->ah;
+
+ /*

```

[patch 04/12] syslets: core code

```
+ * If head does a sys_exit() then the final schedule() must
+ * not be passed on to another cachemiss thread:
+ */
+ t->async_ready = NULL;
+
+ if (unlikely(at))
+   async_thread_exit(at, t);
+
+ if (unlikely(ah))
+   async_head_exit(ah, t);
+}
-
```

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