The utrace User Debugging Infrastructure

The utrace User Debugging Infrastructure

Table of Contents

vi

Chapter 1. utrace concepts

1.1. Introduction

utrace is infrastructure code for tracing and controlling user threads. This is the foundation for writing tracing engines, which can be loadable kernel modules.

The basic actors in utrace are the thread and the tracing engine. A tracing engine is some body of code that calls into the `<linux/utrace.h>` interfaces, represented by a struct utrace_engine_ops. (Usually it's a kernel module, though the legacy `ptrace` support is a tracing engine that is not in a kernel module.) The interface operates on individual threads (struct task_struct). If an engine wants to treat several threads as a group, that is up to its higher-level code.

Tracing begins by attaching an engine to a thread, using `utrace_attach_task` or `utrace_attach_pid`. If successful, it returns a pointer that is the handle used in all other calls.

1.2. Events and Callbacks

An attached engine does nothing by default. An engine makes something happen by requesting callbacks via `utrace_set_events` and poking the thread with `utrace_control`. The synchronization issues related to these two calls are discussed further below in Section 1.4.

Events are specified using the macro `UTRACE_EVENT(`*`type`*`)`. Each event type is associated with a callback in struct utrace_engine_ops. A tracing engine can leave unused callbacks `NULL`. The only callbacks required are those used by the event flags it sets.

Many engines can be attached to each thread. When a thread has an event, each engine gets a callback if it has set the event flag for that event type. For most events, engines are called in the order they attached. Engines that attach after the event has occurred do not get callbacks for that event. This includes any new engines just attached by an existing engine's callback function. Once the sequence of callbacks for that one event has completed, such new engines are then eligible in the next sequence that starts when there is another event.

Event reporting callbacks have details particular to the event type, but are all called in similar environments and have the same constraints. Callbacks are made from safe points, where no locks are held, no special resources are pinned (usually), and the user-mode state of the thread is accessible. So, callback code has a pretty free

hand. But to be a good citizen, callback code should never block for long periods. It is fine to block in `kmalloc` and the like, but never wait for i/o or for user mode to do something. If you need the thread to wait, use `UTRACE_STOP` and return from the callback quickly. When your i/o finishes or whatever, you can use `utrace_control` to resume the thread.

The `UTRACE_EVENT(SYSCALL_ENTRY)` event is a special case. While other events happen in the kernel when it will return to user mode soon, this event happens when entering the kernel before it will proceed with the work requested from user mode. Because of this difference, the `report_syscall_entry` callback is special in two ways. For this event, engines are called in reverse of the normal order (this includes the `report_quiesce` call that precedes a `report_syscall_entry` call). This preserves the semantics that the last engine to attach is called "closest to user mode"--the engine that is first to see a thread's user state when it enters the kernel is also the last to see that state when the thread returns to user mode. For the same reason, if these callbacks use `UTRACE_STOP` (see the next section), the thread stops immediately after callbacks rather than only when it's ready to return to user mode; when allowed to resume, it will actually attempt the system call indicated by the register values at that time.

1.3. Stopping Safely

1.3.1. Writing well-behaved callbacks

Well-behaved callbacks are important to maintain two essential properties of the interface. The first of these is that unrelated tracing engines should not interfere with each other. If your engine's event callback does not return quickly, then another engine won't get the event notification in a timely manner. The second important property is that tracing should be as noninvasive as possible to the normal operation of the system overall and of the traced thread in particular. That is, attached tracing engines should not perturb a thread's behavior, except to the extent that changing its user-visible state is explicitly what you want to do. (Obviously some perturbation is unavoidable, primarily timing changes, ranging from small delays due to the overhead of tracing, to arbitrary pauses in user code execution when a user stops a thread with a debugger for examination.) Even when you explicitly want the perturbation of making the traced thread block, just blocking directly in your callback has more unwanted effects. For example, the `CLONE` event callbacks are called when the new child thread has been created but not yet started running; the child can never be scheduled until the `CLONE` tracing callbacks return. (This allows engines tracing the parent to attach to the child.) If a `CLONE` event callback blocks the parent thread, it also prevents the child thread from running

(even to process a `SIGKILL`). If what you want is to make both the parent and child block, then use `utrace_attach_task` on the child and then use `UTRACE_STOP` on both threads. A more crucial problem with blocking in callbacks is that it can prevent `SIGKILL` from working. A thread that is blocking due to `UTRACE_STOP` will still wake up and die immediately when sent a `SIGKILL`, as all threads should. Relying on the utrace infrastructure rather than on private synchronization calls in event callbacks is an important way to help keep tracing robustly noninvasive.

1.3.2. Using `UTRACE_STOP`

To control another thread and access its state, it must be stopped with `UTRACE_STOP`. This means that it is stopped and won't start running again while we access it. When a thread is not already stopped, `utrace_control` returns `-EINPROGRESS` and an engine must wait for an event callback when the thread is ready to stop. The thread may be running on another CPU or may be blocked. When it is ready to be examined, it will make callbacks to engines that set the `UTRACE_EVENT(QUIESCE)` event bit. To wake up an interruptible wait, use `UTRACE_INTERRUPT`.

As long as some engine has used `UTRACE_STOP` and not called `utrace_control` to resume the thread, then the thread will remain stopped. `SIGKILL` will wake it up, but it will not run user code. When the stop is cleared with `utrace_control` or a callback return value, the thread starts running again. (See also Section 1.4.)

1.4. Tear-down Races

1.4.1. Primacy of `SIGKILL`

Ordinarily synchronization issues for tracing engines are kept fairly straightforward by using `UTRACE_STOP`. You ask a thread to stop, and then once it makes the `report_quiesce` callback it cannot do anything else that would result in another callback, until you let it with a `utrace_control` call. This simple arrangement avoids complex and error-prone code in each one of a tracing engine's event callbacks to keep them serialized with the engine's other operations done on that thread from another thread of control. However, giving tracing engines complete power to keep a traced thread stuck in place runs afoul of a more important kind of simplicity that the kernel overall guarantees: nothing can prevent or delay `SIGKILL` from making a thread die and release its resources. To preserve this important property of `SIGKILL`, it as a special case can break `UTRACE_STOP` like nothing else

normally can. This includes both explicit `SIGKILL` signals and the implicit `SIGKILL` sent to each other thread in the same thread group by a thread doing an exec, or processing a fatal signal, or making an `exit_group` system call. A tracing engine can prevent a thread from beginning the exit or exec or dying by signal (other than `SIGKILL`) if it is attached to that thread, but once the operation begins, no tracing engine can prevent or delay all other threads in the same thread group dying.

1.4.2. Final callbacks

The `report_reap` callback is always the final event in the life cycle of a traced thread. Tracing engines can use this as the trigger to clean up their own data structures. The `report_death` callback is always the penultimate event a tracing engine might see; it's seen unless the thread was already in the midst of dying when the engine attached. Many tracing engines will have no interest in when a parent reaps a dead process, and nothing they want to do with a zombie thread once it dies; for them, the `report_death` callback is the natural place to clean up data structures and detach. To facilitate writing such engines robustly, given the asynchrony of `SIGKILL`, and without error-prone manual implementation of synchronization schemes, the utrace infrastructure provides some special guarantees about the `report_death` and `report_reap` callbacks. It still takes some care to be sure your tracing engine is robust to tear-down races, but these rules make it reasonably straightforward and concise to handle a lot of corner cases correctly.

1.4.3. Engine and task pointers

The first sort of guarantee concerns the core data structures themselves. struct utrace_engine is a reference-counted data structure. While you hold a reference, an engine pointer will always stay valid so that you can safely pass it to any utrace call. Each call to `utrace_attach_task` or `utrace_attach_pid` returns an engine pointer with a reference belonging to the caller. You own that reference until you drop it using `utrace_engine_put`. There is an implicit reference on the engine while it is attached. So if you drop your only reference, and then use `utrace_attach_task` without `UTRACE_ATTACH_CREATE` to look up that same engine, you will get the same pointer with a new reference to replace the one you dropped, just like calling `utrace_engine_get`. When an engine has been detached, either explicitly with `UTRACE_DETACH` or implicitly after `report_reap`, then any references you hold are all that keep the old engine pointer alive.

There is nothing a kernel module can do to keep a struct task_struct alive outside of `rcu_read_lock`. When the task dies and is reaped by its parent (or itself), that structure can be freed so that any dangling pointers you have stored become invalid. utrace will not prevent this, but it can help you detect it safely. By definition, a task

that has been reaped has had all its engines detached. All utrace calls can be safely called on a detached engine if the caller holds a reference on that engine pointer, even if the task pointer passed in the call is invalid. All calls return `-ESRCH` for a detached engine, which tells you that the task pointer you passed could be invalid now. Since `utrace_control` and `utrace_set_events` do not block, you can call those inside a `rcu_read_lock` section and be sure after they don't return `-ESRCH` that the task pointer is still valid until `rcu_read_unlock`. The infrastructure never holds task references of its own. Though neither `rcu_read_lock` nor any other lock is held while making a callback, it's always guaranteed that the struct task_struct and the struct utrace_engine passed as arguments remain valid until the callback function returns.

The common means for safely holding task pointers that is available to kernel modules is to use struct pid, which permits `put_pid` from kernel modules. When using that, the calls `utrace_attach_pid`, `utrace_control_pid`, `utrace_set_events_pid`, and `utrace_barrier_pid` are available.

1.4.4. Serialization of `DEATH` and `REAP`

The second guarantee is the serialization of `DEATH` and `REAP` event callbacks for a given thread. The actual reaping by the parent (`release_task` call) can occur simultaneously while the thread is still doing the final steps of dying, including the `report_death` callback. If a tracing engine has requested both `DEATH` and `REAP` event reports, it's guaranteed that the `report_reap` callback will not be made until after the `report_death` callback has returned. If the `report_death` callback itself detaches from the thread, then the `report_reap` callback will never be made. Thus it is safe for a `report_death` callback to clean up data structures and detach.

1.4.5. Interlock with final callbacks

The final sort of guarantee is that a tracing engine will know for sure whether or not the `report_death` and/or `report_reap` callbacks will be made for a certain thread. These tear-down races are disambiguated by the error return values of `utrace_set_events` and `utrace_control`. Normally `utrace_control` called with `UTRACE_DETACH` returns zero, and this means that no more callbacks will be made. If the thread is in the midst of dying, it returns `-EALREADY` to indicate that the `report_death` callback may already be in progress; when you get this error, you know that any cleanup your `report_death` callback does is about to happen or has just happened--note that if the `report_death` callback does not detach, the engine remains attached until the thread gets reaped. If the thread is in the midst of being reaped, `utrace_control` returns `-ESRCH` to indicate that the `report_reap` callback may already be in progress; this means the engine is implicitly detached

when the callback completes. This makes it possible for a tracing engine that has decided asynchronously to detach from a thread to safely clean up its data structures, knowing that no `report_death` or `report_reap` callback will try to do the same. `utrace_detach` returns `-ESRCH` when the struct utrace_engine has already been detached, but is still a valid pointer because of its reference count. A tracing engine can use this to safely synchronize its own independent multiple threads of control with each other and with its event callbacks that detach.

In the same vein, `utrace_set_events` normally returns zero; if the target thread was stopped before the call, then after a successful call, no event callbacks not requested in the new flags will be made. It fails with `-EALREADY` if you try to clear `UTRACE_EVENT(DEATH)` when the `report_death` callback may already have begun, if you try to clear `UTRACE_EVENT(REAP)` when the `report_reap` callback may already have begun, or if you try to newly set `UTRACE_EVENT(DEATH)` or `UTRACE_EVENT(QUIESCE)` when the target is already dead or dying. Like `utrace_control`, it returns `-ESRCH` when the thread has already been detached (including forcible detach on reaping). This lets the tracing engine know for sure which event callbacks it will or won't see after `utrace_set_events` has returned. By checking for errors, it can know whether to clean up its data structures immediately or to let its callbacks do the work.

1.4.6. Using `utrace_barrier`

When a thread is safely stopped, calling `utrace_control` with `UTRACE_DETACH` or calling `utrace_set_events` to disable some events ensures synchronously that your engine won't get any more of the callbacks that have been disabled (none at all when detaching). But these can also be used while the thread is not stopped, when it might be simultaneously making a callback to your engine. For this situation, these calls return `-EINPROGRESS` when it's possible a callback is in progress. If you are not prepared to have your old callbacks still run, then you can synchronize to be sure all the old callbacks are finished, using `utrace_barrier`. This is necessary if the kernel module containing your callback code is going to be unloaded.

After using `UTRACE_DETACH` once, further calls to `utrace_control` with the same engine pointer will return `-ESRCH`. In contrast, after getting `-EINPROGRESS` from `utrace_set_events`, you can call `utrace_set_events` again later and if it returns zero then know the old callbacks have finished.

Unlike all other calls, `utrace_barrier` (and `utrace_barrier_pid`) will accept any engine pointer you hold a reference on, even if `UTRACE_DETACH` has already been used. After any `utrace_control` or `utrace_set_events` call (these do not block), you can call `utrace_barrier` to block until callbacks have finished. This returns `-ESRCH` only if the engine is completely detached (finished all callbacks). Otherwise it waits until the thread is definitely not in the midst of a callback to this

engine and then returns zero, but can return `-ERESTARTSYS` if its wait is interrupted.

Chapter 2. utrace core API

The utrace API is declared in `<linux/utrace.h>`.

enum utrace_resume_action

Name

`enum utrace_resume_action` — engine's choice of action for a traced task

Synopsis

```
enum utrace_resume_action {
  UTRACE_STOP,
  UTRACE_INTERRUPT,
  UTRACE_REPORT,
  UTRACE_SINGLESTEP,
  UTRACE_BLOCKSTEP,
  UTRACE_RESUME,
  UTRACE_DETACH
};
```

Constants

UTRACE_STOP

Stay quiescent after callbacks.

UTRACE_INTERRUPT

Make *report_signal*() callback soon.

UTRACE_REPORT

Make some callback soon.

UTRACE_SINGLESTEP

Resume in user mode for one instruction.

UTRACE_BLOCKSTEP

Resume in user mode until next branch.

UTRACE_RESUME

Resume normally in user mode.

UTRACE_DETACH

Detach my engine (implies UTRACE_RESUME).

Description

See utrace_control for detailed descriptions of each action. This is encoded in the *action* argument and the return value for every callback with a u32 return value.

The order of these is important. When there is more than one engine, each supplies its choice and the smallest value prevails.

utrace_resume_action

LINUX

Kernel Hackers ManualJuly 2010

Name

utrace_resume_action — enum utrace_resume_action from callback action

Synopsis

enum utrace_resume_action **utrace_resume_action** (u32 *action*);

Arguments

`action`

> u32 callback `action` argument or return value

Description

This extracts the enum utrace_resume_action from `action`, which is the `action` argument to a struct utrace_engine_ops callback or the return value from one.

enum utrace_signal_action

Name

`enum utrace_signal_action` — disposition of signal

Synopsis

```
enum utrace_signal_action {
  UTRACE_SIGNAL_DELIVER,
  UTRACE_SIGNAL_IGN,
  UTRACE_SIGNAL_TERM,
  UTRACE_SIGNAL_CORE,
  UTRACE_SIGNAL_STOP,
  UTRACE_SIGNAL_TSTP,
  UTRACE_SIGNAL_REPORT,
  UTRACE_SIGNAL_HANDLER
};
```

Constants

UTRACE_SIGNAL_DELIVER

> Deliver according to sigaction.

UTRACE_SIGNAL_IGN

> Ignore the signal.

UTRACE_SIGNAL_TERM

> Terminate the process.

UTRACE_SIGNAL_CORE

> Terminate with core dump.

UTRACE_SIGNAL_STOP

> Deliver as absolute stop.

UTRACE_SIGNAL_TSTP

> Deliver as job control stop.

UTRACE_SIGNAL_REPORT

> Reporting before pending signals.

UTRACE_SIGNAL_HANDLER

> Reporting after signal handler setup.

Description

This is encoded in the `action` argument and the return value for a `report_signal`() callback. It says what will happen to the signal described by the siginfo_t parameter to the callback.

The UTRACE_SIGNAL_REPORT value is used in an `action` argument when a tracing report is being made before dequeuing any pending signal. If this is immediately after a signal handler has been set up, then UTRACE_SIGNAL_HANDLER is used instead. A `report_signal` callback that uses UTRACE_SIGNAL_DELIVER|UTRACE_SINGLESTEP will ensure it sees a UTRACE_SIGNAL_HANDLER report.

utrace_signal_action

Name

`utrace_signal_action` — enum utrace_signal_action from callback action

Synopsis

```
enum utrace_signal_action utrace_signal_action (u32 action);
```

Arguments

`action`

> `report_signal` callback `action` argument or return value

Description

This extracts the enum utrace_signal_action from `action`, which is the `action` argument to a `report_signal` callback or the return value from one.

enum utrace_syscall_action

Name

`enum utrace_syscall_action` — disposition of system call attempt

Synopsis

```
enum utrace_syscall_action {
  UTRACE_SYSCALL_RUN,
  UTRACE_SYSCALL_ABORT
};
```

Constants

UTRACE_SYSCALL_RUN

Run the system call.

UTRACE_SYSCALL_ABORT

Don't run the system call.

Description

This is encoded in the *action* argument and the return value for a *report_syscall_entry* callback.

utrace_syscall_action

Name

`utrace_syscall_action` — enum utrace_syscall_action from callback action

Synopsis

```
enum utrace_syscall_action utrace_syscall_action (u32 action);
```

Arguments

action

> `report_syscall_entry` callback *action* or return value

Description

This extracts the enum utrace_syscall_action from `action`, which is the `action` argument to a `report_syscall_entry` callback or the return value from one.

struct utrace_engine

Name

struct utrace_engine — per-engine structure

Synopsis

```
struct utrace_engine {
  const struct utrace_engine_ops * ops;
  void * data;
  unsigned long flags;
};
```

Members

ops

> struct utrace_engine_ops pointer passed to `utrace_attach_task`

data

> engine-private void * passed to `utrace_attach_task`

flags

> event mask set by `utrace_set_events` plus internal flag bits

Description

The task itself never has to worry about engines detaching while it's doing event callbacks. These structures are removed from the task's active list only when it's stopped, or by the task itself.

`utrace_engine_get` and `utrace_engine_put` maintain a reference count. When it drops to zero, the structure is freed. One reference is held implicitly while the engine is attached to its task.

utrace_engine_get

Name

`utrace_engine_get` — acquire a reference on a struct utrace_engine

Synopsis

```
void utrace_engine_get (struct utrace_engine * engine);
```

Arguments

engine

> struct utrace_engine pointer

Description

You must hold a reference on *engine*, and you get another.

utrace_engine_put

Name

utrace_engine_put — release a reference on a struct utrace_engine

Synopsis

void **utrace_engine_put** (struct utrace_engine * *engine*);

Arguments

engine

> struct utrace_engine pointer

Description

You must hold a reference on `engine`, and you lose that reference. If it was the last one, `engine` becomes an invalid pointer.

struct utrace_engine_ops

Name

struct utrace_engine_ops — tracing engine callbacks

Synopsis

```
struct utrace_engine_ops {
  u32 (* report_quiesce) (enum utrace_resume_action action,struct utrace_e
  u32 (* report_signal) (u32 action,struct utrace_engine *engine,struct ta
  u32 (* report_clone) (enum utrace_resume_action action,struct utrace_eng
  u32 (* report_jctl) (enum utrace_resume_action action,struct utrace_engi
  u32 (* report_exec) (enum utrace_resume_action action,struct utrace_engi
  u32 (* report_syscall_entry) (u32 action,struct utrace_engine *engine,st
  u32 (* report_syscall_exit) (enum utrace_resume_action action,struct utr
  u32 (* report_exit) (enum utrace_resume_action action,struct utrace_engi
  u32 (* report_death) (struct utrace_engine *engine,struct task_struct *t
  void (* report_reap) (struct utrace_engine *engine,struct task_struct *t
  void (* release) (void *data);
};
```

Members

report_quiesce

> Requested by `UTRACE_EVENT(QUIESCE)`. This does not indicate any event, but just that `task` (the current thread) is in a safe place for examination. This call is made before each specific event callback, except for `report_reap`. The

`event` argument gives the `UTRACE_EVENT(`which`)` value for the event occurring. This callback might be made for events `engine` has not requested, if some other engine is tracing the event; calling `utrace_set_events` call here can request the immediate callback for this occurrence of `event`. `event` is zero when there is no other event, `task` is now ready to check for signals and return to user mode, and some engine has used `UTRACE_REPORT` or `UTRACE_INTERRUPT` to request this callback. For this case, if `report_signal` is not `NULL`, the `report_quiesce` callback may be replaced with a `report_signal` callback passing `UTRACE_SIGNAL_REPORT` in its `action` argument, whenever `task` is entering the signal-check path anyway.

report_signal

Requested by `UTRACE_EVENT(SIGNAL_*)` or `UTRACE_EVENT(QUIESCE)`. Use `utrace_signal_action` and `utrace_resume_action` on `action`. The signal action is `UTRACE_SIGNAL_REPORT` when some engine has used `UTRACE_REPORT` or `UTRACE_INTERRUPT`; the callback can choose to stop or to deliver an artificial signal, before pending signals. It's `UTRACE_SIGNAL_HANDLER` instead when signal handler setup just finished (after a previous `UTRACE_SIGNAL_DELIVER` return); this serves in lieu of any `UTRACE_SIGNAL_REPORT` callback requested by `UTRACE_REPORT` or `UTRACE_INTERRUPT`, and is also implicitly requested by `UTRACE_SINGLESTEP` or `UTRACE_BLOCKSTEP` into the signal delivery. The other signal actions indicate a signal about to be delivered; the previous engine's return value sets the signal action seen by the the following engine's callback. The `info` data can be changed at will, including `info`->si_signo. The settings in `return_ka` determines what `UTRACE_SIGNAL_DELIVER` does. `orig_ka` is what was in force before other tracing engines intervened, and it's `NULL` when this report began as `UTRACE_SIGNAL_REPORT` or `UTRACE_SIGNAL_HANDLER`. For a report without a new signal, `info` is left uninitialized and must be set completely by an engine that chooses to deliver a signal; if there was a previous `report_signal` callback ending in `UTRACE_STOP` and it was just resumed using `UTRACE_REPORT` or `UTRACE_INTERRUPT`, then `info` is left unchanged from the previous callback. In this way, the original signal can be left in `info` while returning `UTRACE_STOP|UTRACE_SIGNAL_IGN` and then found again when resuming `task` with `UTRACE_INTERRUPT`. The `UTRACE_SIGNAL_HOLD` flag bit can be OR'd into the return value, and might be in `action` if the previous engine returned it. This flag asks that the signal in `info` be pushed back on `task`'s queue so that it will be seen again after whatever action is taken now.

report_clone

Requested by `UTRACE_EVENT(CLONE)`. Event reported for parent, before the new task `child` might run. `clone_flags` gives the flags used in the clone

system call, or equivalent flags for a `fork` or `vfork` system call. This function can use `utrace_attach_task` on *child*. It's guaranteed that asynchronous `utrace_attach_task` calls will be ordered after any calls in *report_clone* callbacks for the parent. Thus when using `UTRACE_ATTACH_EXCLUSIVE` in the asynchronous calls, you can be sure that the parent's *report_clone* callback has already attached to *child* or chosen not to. Passing `UTRACE_STOP` to `utrace_control` on *child* here keeps the child stopped before it ever runs in user mode, `UTRACE_REPORT` or `UTRACE_INTERRUPT` ensures a callback from *child* before it starts in user mode.

report_jctl

Requested by `UTRACE_EVENT(JCTL)`. Job control event; *type* is `CLD_STOPPED` or `CLD_CONTINUED`, indicating whether we are stopping or resuming now. If *notify* is nonzero, *task* is the last thread to stop and so will send `SIGCHLD` to its parent after this callback; *notify* reflects what the parent's `SIGCHLD` has in *si_code*, which can sometimes be `CLD_STOPPED` even when *type* is `CLD_CONTINUED`.

report_exec

Requested by `UTRACE_EVENT(EXEC)`. An execve system call has succeeded and the new program is about to start running. The initial user register state is handy to be tweaked directly in *regs*. *fmt* and *bprm* gives the details of this exec.

report_syscall_entry

Requested by `UTRACE_EVENT(SYSCALL_ENTRY)`. Thread has entered the kernel to request a system call. The user register state is handy to be tweaked directly in *regs*. The *action* argument contains an enum utrace_syscall_action, use `utrace_syscall_action` to extract it. The return value overrides the last engine's action for the system call. If the final action is `UTRACE_SYSCALL_ABORT`, no system call is made. The details of the system call being attempted can be fetched here with `syscall_get_nr` and `syscall_get_arguments`. The parameter registers can be changed with `syscall_set_arguments`.

report_syscall_exit

Requested by `UTRACE_EVENT(SYSCALL_EXIT)`. Thread is about to leave the kernel after a system call request. The user register state is handy to be tweaked directly in *regs*. The results of the system call attempt can be examined here using `syscall_get_error` and `syscall_get_return_value`. It is safe here to call `syscall_set_return_value` or `syscall_rollback`.

report_exit

> Requested by UTRACE_EVENT(EXIT). Thread is exiting and cannot be prevented from doing so, but all its state is still live. The *code* value will be the wait result seen by the parent, and can be changed by this engine or others. The *orig_code* value is the real status, not changed by any tracing engine. Returning UTRACE_STOP here keeps *task* stopped before it cleans up its state and dies, so it can be examined by other processes. When *task* is allowed to run, it will die and get to the *report_death* callback.

report_death

> Requested by UTRACE_EVENT(DEATH). Thread is really dead now. It might be reaped by its parent at any time, or self-reap immediately. Though the actual reaping may happen in parallel, a report_reap callback will always be ordered after a report_death callback.

report_reap

> Requested by UTRACE_EVENT(REAP). Called when someone reaps the dead task (parent, init, or self). This means the parent called wait, or else this was a detached thread or a process whose parent ignores SIGCHLD. No more callbacks are made after this one. The engine is always detached. There is nothing more a tracing engine can do about this thread. After this callback, the *engine* pointer will become invalid. The *task* pointer may become invalid if get_task_struct hasn't been used to keep it alive. An engine should always request this callback if it stores the *engine* pointer or stores any pointer in *engine*->data, so it can clean up its data structures. Unlike other callbacks, this can be called from the parent's context rather than from the traced thread itself--it must not delay the parent by blocking.

release

> If not NULL, this is called after the last utrace_engine_put call for a struct utrace_engine, which could be implicit after a UTRACE_DETACH return from another callback. Its argument is the engine's *data* member.

Description

Each *report_*()* callback corresponds to an UTRACE_EVENT(*) bit. utrace_set_events calls on *engine* choose which callbacks will be made to *engine* from *task*.

Most callbacks take an *action* argument, giving the resume action chosen by other tracing engines. All callbacks take an *engine* argument, and a *task* argument,

which is always equal to `current`. For some calls, `action` also includes bits specific to that event and `utrace_resume_action` is used to extract the resume action. This shows what would happen if `engine` wasn't there, or will if the callback's return value uses `UTRACE_RESUME`. This always starts as `UTRACE_RESUME` when no other tracing is being done on this task.

All return values contain enum utrace_resume_action bits. For some calls, other bits specific to that kind of event are added to the resume action bits with OR. These are the same bits used in the `action` argument. The resume action returned by a callback does not override previous engines' choices, it only says what `engine` wants done. What `task` actually does is the action that's most constrained among the choices made by all attached engines. See `utrace_control` for more information on the actions.

When `UTRACE_STOP` is used in `report_syscall_entry`, then `task` stops before attempting the system call. In other cases, the resume action does not take effect until `task` is ready to check for signals and return to user mode. If there are more callbacks to be made, the last round of calls determines the final action. A `report_quiesce` callback with `event` zero, or a `report_signal` callback, will always be the last one made before `task` resumes. Only `UTRACE_STOP` is "sticky"--if `engine` returned `UTRACE_STOP` then `task` stays stopped unless `engine` returns different from a following callback.

The `report_death` and `report_reap` callbacks do not take `action` arguments, and only `UTRACE_DETACH` is meaningful in the return value from a `report_death` callback. None of the resume actions applies to a dead thread.

All `report_*()` hooks are called with no locks held, in a generally safe environment when we will be returning to user mode soon (or just entered the kernel). It is fine to block for memory allocation and the like, but all hooks are asynchronous and must not block on external events! If you want the thread to block, use `UTRACE_STOP` in your hook's return value; then later wake it up with `utrace_control`.

struct utrace_examiner

LINUX

Name

`struct utrace_examiner` — private state for using `utrace_prepare_examine`

Synopsis

```
struct utrace_examiner {
};
```

Members

None

Description

The members of struct utrace_examiner are private to the implementation. This data type holds the state from a call to `utrace_prepare_examine` to be used by a call to `utrace_finish_examine`.

utrace_control_pid

Name

`utrace_control_pid` — control a thread being traced by a tracing engine

Synopsis

```
__must_check int utrace_control_pid (struct pid * pid, struct
utrace_engine * engine, enum utrace_resume_action action);
```

Arguments

pid

    thread to affect

engine

    attached engine to affect

action

    enum utrace_resume_action for thread to do

Description

This is the same as `utrace_control`, but takes a struct pid pointer rather than a struct task_struct pointer. The caller must hold a ref on `pid`, but does not need to worry about the task staying valid. If it's been reaped so that `pid` points nowhere, then this call returns `-ESRCH`.

utrace_set_events_pid

Name

`utrace_set_events_pid` — choose which event reports a tracing engine gets

Synopsis

`__must_check int` **`utrace_set_events_pid`** `(struct pid *` *`pid`*`,`
`struct utrace_engine *` *`engine`*`, unsigned long` *`eventmask`*`);`

Arguments

`pid`

> thread to affect

`engine`

> attached engine to affect

`eventmask`

> new event mask

Description

This is the same as `utrace_set_events`, but takes a struct pid pointer rather than a struct task_struct pointer. The caller must hold a ref on *`pid`*, but does not need to worry about the task staying valid. If it's been reaped so that *`pid`* points nowhere, then this call returns -`ESRCH`.

utrace_barrier_pid

LINUX

Kernel Hackers ManualJuly 2010

Name

`utrace_barrier_pid` — synchronize with simultaneous tracing callbacks

Synopsis

```
__must_check int utrace_barrier_pid (struct pid * pid, struct
utrace_engine * engine);
```

Arguments

pid

　　thread to affect

engine

　　engine to affect (can be detached)

Description

This is the same as `utrace_barrier`, but takes a struct pid pointer rather than a struct task_struct pointer. The caller must hold a ref on `pid`, but does not need to worry about the task staying valid. If it's been reaped so that `pid` points nowhere, then this call returns `-ESRCH`.

utrace_attach_task

LINUX

Kernel Hackers ManualJuly 2010

Name

utrace_attach_task — attach new engine, or look up an attached engine

Synopsis

```
struct utrace_engine * utrace_attach_task (struct task_struct
* target, int flags, const struct utrace_engine_ops * ops,
void * data);
```

Arguments

target

    thread to attach to

flags

    flag bits combined with OR, see below

ops

    callback table for new engine

data

    engine private data pointer

Description

The caller must ensure that the `target` thread does not get freed, i.e. hold a ref or be its parent. It is always safe to call this on `current`, or on the `child` pointer in a `report_clone` callback. For most other cases, it's easier to use `utrace_attach_pid` instead.

UTRACE_ATTACH_CREATE

Create a new engine. If `UTRACE_ATTACH_CREATE` is not specified, you only look up an existing engine already attached to the thread.

UTRACE_ATTACH_EXCLUSIVE

Attempting to attach a second (matching) engine fails with `-EEXIST`.

UTRACE_ATTACH_MATCH_OPS

Only consider engines matching `ops`.

UTRACE_ATTACH_MATCH_DATA

Only consider engines matching `data`.

Calls with neither `UTRACE_ATTACH_MATCH_OPS` nor `UTRACE_ATTACH_MATCH_DATA` match the first among any engines attached to `target`. That means that `UTRACE_ATTACH_EXCLUSIVE` in such a call fails with `-EEXIST` if there are any engines on `target` at all.

utrace_attach_pid

LINUX

Kernel Hackers ManualJuly 2010

Name

`utrace_attach_pid` — attach new engine, or look up an attached engine

Synopsis

```
struct utrace_engine * utrace_attach_pid (struct pid * pid,
int flags, const struct utrace_engine_ops * ops, void * data);
```

Arguments

`pid`

  struct pid pointer representing thread to attach to

flags

flag bits combined with OR, see `utrace_attach_task`

ops

callback table for new engine

data

engine private data pointer

Description

This is the same as `utrace_attach_task`, but takes a struct pid pointer rather than a struct task_struct pointer. The caller must hold a ref on *pid*, but does not need to worry about the task staying valid. If it's been reaped so that *pid* points nowhere, then this call returns -`ESRCH`.

utrace_set_events

Name

`utrace_set_events` — choose which event reports a tracing engine gets

Synopsis

```
int utrace_set_events (struct task_struct * target, struct
utrace_engine * engine, unsigned long events);
```

Arguments

target

    thread to affect

engine

    attached engine to affect

events

    new event mask

Description

This changes the set of events for which `engine` wants callbacks made.

This fails with -EALREADY and does nothing if you try to clear UTRACE_EVENT(DEATH) when the `report_death` callback may already have begun, if you try to clear UTRACE_EVENT(REAP) when the `report_reap` callback may already have begun, or if you try to newly set UTRACE_EVENT(DEATH) or UTRACE_EVENT(QUIESCE) when `target` is already dead or dying.

This can fail with -ESRCH when `target` has already been detached, including forcible detach on reaping.

If `target` was stopped before the call, then after a successful call, no event callbacks not requested in `events` will be made; if UTRACE_EVENT(QUIESCE) is included in `events`, then a `report_quiesce` callback will be made when `target` resumes.

If `target` was not stopped and `events` excludes some bits that were set before, this can return -EINPROGRESS to indicate that `target` may have been making some callback to `engine`. When this returns zero, you can be sure that no event callbacks you've disabled in `events` can be made. If `events` only sets new bits that were not set before on `engine`, then -EINPROGRESS will never be returned.

To synchronize after an -EINPROGRESS return, see `utrace_barrier`.

When `target` is `current`, -EINPROGRESS is not returned. But note that a newly-created engine will not receive any callbacks related to an event notification already in progress. This call enables `events` callbacks to be made as soon as `engine` becomes eligible for any callbacks, see `utrace_attach_task`.

These rules provide for coherent synchronization based on UTRACE_STOP, even when SIGKILL is breaking its normal simple rules.

utrace_control

Name

utrace_control — control a thread being traced by a tracing engine

Synopsis

```
int utrace_control (struct task_struct * target, struct
utrace_engine * engine, enum utrace_resume_action action);
```

Arguments

target

    thread to affect

engine

    attached engine to affect

action

    enum utrace_resume_action for thread to do

Description

This is how a tracing engine asks a traced thread to do something. This call is controlled by the *action* argument, which has the same meaning as the enum utrace_resume_action value returned by event reporting callbacks.

If `target` is already dead (`target`->exit_state nonzero), all actions except UTRACE_DETACH fail with -ESRCH.

The following sections describe each option for the `action` argument.

UTRACE_DETACH

After this, the `engine` data structure is no longer accessible, and the thread might be reaped. The thread will start running again if it was stopped and no longer has any attached engines that want it stopped.

If the `report_reap` callback may already have begun, this fails with -ESRCH. If the `report_death` callback may already have begun, this fails with -EALREADY.

If `target` is not already stopped, then a callback to this engine might be in progress or about to start on another CPU. If so, then this returns -EINPROGRESS; the detach happens as soon as the pending callback is finished. To synchronize after an -EINPROGRESS return, see `utrace_barrier`.

If `target` is properly stopped before `utrace_control` is called, then after successful return it's guaranteed that no more callbacks to the `engine`->ops vector will be made.

The only exception is SIGKILL (and exec or group-exit by another thread in the group), which can cause asynchronous `report_death` and/or `report_reap` callbacks even when UTRACE_STOP was used. (In that event, this fails with -ESRCH or -EALREADY, see above.)

UTRACE_STOP

This asks that `target` stop running. This returns 0 only if `target` is already stopped, either for tracing or for job control. Then `target` will remain stopped until another `utrace_control` call is made on `engine`; `target` can be woken only by SIGKILL (or equivalent, such as exec or termination by another thread in the same thread group).

This returns -EINPROGRESS if `target` is not already stopped. Then the effect is like UTRACE_REPORT. A `report_quiesce` or `report_signal` callback will be made soon. Your callback can then return UTRACE_STOP to keep `target` stopped.

This does not interrupt system calls in progress, including ones that sleep for a long time. For that, use UTRACE_INTERRUPT. To interrupt system calls and then keep `target` stopped, your `report_signal` callback can return UTRACE_STOP.

UTRACE_RESUME

Just let `target` continue running normally, reversing the effect of a previous `UTRACE_STOP`. If another engine is keeping `target` stopped, then it remains stopped until all engines let it resume. If `target` was not stopped, this has no effect.

UTRACE_REPORT

This is like `UTRACE_RESUME`, but also ensures that there will be a `report_quiesce` or `report_signal` callback made soon. If `target` had been stopped, then there will be a callback before it resumes running normally. If another engine is keeping `target` stopped, then there might be no callbacks until all engines let it resume.

Since this is meaningless unless `report_quiesce` callbacks will be made, it returns -`EINVAL` if `engine` lacks `UTRACE_EVENT(QUIESCE)`.

UTRACE_INTERRUPT

This is like `UTRACE_REPORT`, but ensures that `target` will make a `report_signal` callback before it resumes or delivers signals. If `target` was in a system call or about to enter one, work in progress will be interrupted as if by `SIGSTOP`. If another engine is keeping `target` stopped, then there might be no callbacks until all engines let it resume.

This gives `engine` an opportunity to introduce a forced signal disposition via its `report_signal` callback.

UTRACE_SINGLESTEP

It's invalid to use this unless `arch_has_single_step` returned true. This is like `UTRACE_RESUME`, but resumes for one user instruction only. It's invalid to use this in `utrace_control` unless `target` had been stopped by `engine` previously.

Note that passing `UTRACE_SINGLESTEP` or `UTRACE_BLOCKSTEP` to `utrace_control` or returning it from an event callback alone does not necessarily ensure that stepping will be enabled. If there are more callbacks made to any engine before returning to user mode, then the resume action is chosen only by the last set of callbacks. To be sure, enable `UTRACE_EVENT(QUIESCE)` and look for the `report_quiesce` callback with a zero event mask, or the `report_signal` callback with `UTRACE_SIGNAL_REPORT`.

Since this is not robust unless *report_quiesce* callbacks will be made, it returns -EINVAL if *engine* lacks UTRACE_EVENT(QUIESCE).

UTRACE_BLOCKSTEP

It's invalid to use this unless arch_has_block_step returned true. This is like UTRACE_SINGLESTEP, but resumes for one whole basic block of user instructions.

Since this is not robust unless *report_quiesce* callbacks will be made, it returns -EINVAL if *engine* lacks UTRACE_EVENT(QUIESCE).

UTRACE_BLOCKSTEP devolves to UTRACE_SINGLESTEP when another tracing engine is using UTRACE_SINGLESTEP at the same time.

utrace_barrier

LINUX

Kernel Hackers ManualJuly 2010

Name

utrace_barrier — synchronize with simultaneous tracing callbacks

Synopsis

```
int utrace_barrier (struct task_struct * target, struct
utrace_engine * engine);
```

Arguments

target

thread to affect

engine

> engine to affect (can be detached)

Description

This blocks while *target* might be in the midst of making a callback to *engine*. It can be interrupted by signals and will return -ERESTARTSYS. A return value of zero means no callback from *target* to *engine* was in progress. Any effect of its return value (such as UTRACE_STOP) has already been applied to *engine*.

It's not necessary to keep the *target* pointer alive for this call. It's only necessary to hold a ref on *engine*. This will return safely even if *target* has been reaped and has no task refs.

A successful return from utrace_barrier guarantees its ordering with respect to utrace_set_events and utrace_control calls. If *target* was not properly stopped, event callbacks just disabled might still be in progress; utrace_barrier waits until there is no chance an unwanted callback can be in progress.

utrace_prepare_examine

Name

utrace_prepare_examine — prepare to examine thread state

Synopsis

```
int utrace_prepare_examine (struct task_struct * target,
struct utrace_engine * engine, struct utrace_examiner * exam);
```

Arguments

target

> thread of interest, a struct task_struct pointer

engine

> engine pointer returned by `utrace_attach_task`

exam

> temporary state, a struct utrace_examiner pointer

Description

This call prepares to safely examine the thread *target* using struct user_regset calls, or direct access to thread-synchronous fields.

When *target* is current, this call is superfluous. When *target* is another thread, it must held stopped via `UTRACE_STOP` by *engine*.

This call may block the caller until *target* stays stopped, so it must be called only after the caller is sure *target* is about to unschedule. This means a zero return from a `utrace_control` call on *engine* giving `UTRACE_STOP`, or a `report_quiesce` or `report_signal` callback to *engine* that used `UTRACE_STOP` in its return value.

Returns -`ESRCH` if *target* is dead or -`EINVAL` if `UTRACE_STOP` was not used. If *target* has started running again despite `UTRACE_STOP` (for `SIGKILL` or a spurious wakeup), this call returns -`EAGAIN`.

When this call returns zero, it's safe to use struct user_regset calls and `task_user_regset_view` on *target* and to examine some of its fields directly. When the examination is complete, a `utrace_finish_examine` call must follow to check whether it was completed safely.

utrace_finish_examine

LINUX

Name

`utrace_finish_examine` — complete an examination of thread state

Synopsis

```
int utrace_finish_examine (struct task_struct * target, struct
utrace_engine * engine, struct utrace_examiner * exam);
```

Arguments

target

thread of interest, a struct task_struct pointer

engine

engine pointer returned by `utrace_attach_task`

exam

pointer passed to `utrace_prepare_examine` call

Description

This call completes an examination on the thread `target` begun by a paired
`utrace_prepare_examine` call with the same arguments that returned success
(zero).

When `target` is current, this call is superfluous. When `target` is another thread,
this returns zero if `target` has remained unscheduled since the paired
`utrace_prepare_examine` call returned zero.

When this returns an error, any examination done since the paired
`utrace_prepare_examine` call is unreliable and the data extracted should be
discarded. The error is -`EINVAL` if `engine` is not keeping `target` stopped, or
-`EAGAIN` if `target` woke up unexpectedly.

Chapter 3. Machine State

The `task_current_syscall` function can be used on any valid struct task_struct at any time, and does not even require that `utrace_attach_task` was used at all.

The other ways to access the registers and other machine-dependent state of a task can only be used on a task that is at a known safe point. The safe points are all the places where `utrace_set_events` can request callbacks (except for the `DEATH` and `REAP` events). So at any event callback, it is safe to examine `current`.

One task can examine another only after a callback in the target task that returns `UTRACE_STOP` so that task will not return to user mode after the safe point. This guarantees that the task will not resume until the same engine uses `utrace_control`, unless the task dies suddenly. To examine safely, one must use a pair of calls to `utrace_prepare_examine` and `utrace_finish_examine` surrounding the calls to struct user_regset functions or direct examination of task data structures. `utrace_prepare_examine` returns an error if the task is not properly stopped and not dead. After a successful examination, the paired `utrace_finish_examine` call returns an error if the task ever woke up during the examination. If so, any data gathered may be scrambled and should be discarded. This means there was a spurious wake-up (which should not happen), or a sudden death.

3.1. struct user_regset

The struct user_regset API is declared in `<linux/regset.h>`.

user_regset_active_fn

Name

`user_regset_active_fn` — type of *active* function in struct user_regset

Synopsis

```
typedef int user_regset_active_fn (struct task_struct *
target, const struct user_regset * regset);
```

Arguments

target

> thread being examined

regset

> regset being examined

Description

Return -ENODEV if not available on the hardware found. Return 0 if no interesting state in this thread. Return >0 number of *size* units of interesting state. Any get call fetching state beyond that number will see the default initialization state for this data, so a caller that knows what the default state is need not copy it all out. This call is optional; the pointer is NULL if there is no inexpensive check to yield a value < *n*.

user_regset_get_fn

Name

user_regset_get_fn — type of *get* function in struct user_regset

Synopsis

```
typedef int user_regset_get_fn (struct task_struct * target,
const struct user_regset * regset, unsigned int pos, unsigned
int count, void * kbuf, void __user * ubuf);
```

Arguments

target

    thread being examined

regset

    regset being examined

pos

    offset into the regset data to access, in bytes

count

    amount of data to copy, in bytes

kbuf

    if not NULL, a kernel-space pointer to copy into

ubuf

    if *kbuf* is NULL, a user-space pointer to copy into

Description

Fetch register values. Return 0 on success; -EIO or -ENODEV are usual failure returns. The *pos* and *count* values are in bytes, but must be properly aligned. If *kbuf* is non-null, that buffer is used and *ubuf* is ignored. If *kbuf* is NULL, then ubuf gives a userland pointer to access directly, and an -EFAULT return value is possible.

user_regset_set_fn

Name

`user_regset_set_fn` — type of `set` function in struct user_regset

Synopsis

```
typedef int user_regset_set_fn (struct task_struct * target,
const struct user_regset * regset, unsigned int pos, unsigned
int count, const void * kbuf, const void __user * ubuf);
```

Arguments

`target`

   thread being examined

`regset`

   regset being examined

`pos`

   offset into the regset data to access, in bytes

`count`

   amount of data to copy, in bytes

`kbuf`

   if not NULL, a kernel-space pointer to copy from

`ubuf`

   if `kbuf` is NULL, a user-space pointer to copy from

Description

Store register values. Return `0` on success; `-EIO` or `-ENODEV` are usual failure returns. The `pos` and `count` values are in bytes, but must be properly aligned. If `kbuf` is non-null, that buffer is used and `ubuf` is ignored. If `kbuf` is `NULL`, then ubuf gives a userland pointer to access directly, and an `-EFAULT` return value is possible.

user_regset_writeback_fn

Name

`user_regset_writeback_fn` — type of `writeback` function in struct user_regset

Synopsis

```
typedef int user_regset_writeback_fn (struct task_struct *
target, const struct user_regset * regset, int immediate);
```

Arguments

`target`

thread being examined

`regset`

regset being examined

`immediate`

zero if writeback at completion of next context switch is OK

Description

This call is optional; usually the pointer is `NULL`. When provided, there is some user memory associated with this regset's hardware, such as memory backing cached register data on register window machines; the regset's data controls what user memory is used (e.g. via the stack pointer value).

Write register data back to user memory. If the `immediate` flag is nonzero, it must be written to the user memory so uaccess or `access_process_vm` can see it when this call returns; if zero, then it must be written back by the time the task completes a context switch (as synchronized with `wait_task_inactive`). Return `0` on success or if there was nothing to do, `-EFAULT` for a memory problem (bad stack pointer or whatever), or `-EIO` for a hardware problem.

struct user_regset

Name

`struct user_regset` — accessible thread CPU state

Synopsis

```
struct user_regset {
  user_regset_get_fn * get;
  user_regset_set_fn * set;
  user_regset_active_fn * active;
  user_regset_writeback_fn * writeback;
  unsigned int n;
  unsigned int size;
  unsigned int align;
  unsigned int bias;
  unsigned int core_note_type;
};
```

Members

get

> Function to fetch values.

set

> Function to store values.

active

> Function to report if regset is active, or `NULL`.

writeback

> Function to write data back to user memory, or `NULL`.

n

> Number of slots (registers).

size

> Size in bytes of a slot (register).

align

> Required alignment, in bytes.

bias

> Bias from natural indexing.

core_note_type

> ELF note `n_type` value used in core dumps.

Description

This data structure describes a machine resource we call a register set. This is part of the state of an individual thread, not necessarily actual CPU registers per se. A register set consists of a number of similar slots, given by n. Each slot is `size` bytes, and aligned to `align` bytes (which is at least `size`).

These functions must be called only on the current thread or on a thread that is in `TASK_STOPPED` or `TASK_TRACED` state, that we are guaranteed will not be woken up and return to user mode, and that we have called `wait_task_inactive` on. (The target thread always might wake up for SIGKILL while these functions are working, in which case that thread's user_regset state might be scrambled.)

The `pos` argument must be aligned according to `align`; the `count` argument must be a multiple of `size`. These functions are not responsible for checking for invalid arguments.

When there is a natural value to use as an index, `bias` gives the difference between the natural index and the slot index for the register set. For example, x86 GDT segment descriptors form a regset; the segment selector produces a natural index, but only a subset of that index space is available as a regset (the TLS slots); subtracting `bias` from a segment selector index value computes the regset slot.

If nonzero, `core_note_type` gives the n_type field (NT_* value) of the core file note in which this regset's data appears. NT_PRSTATUS is a special case in that the regset data starts at offsetof(struct elf_prstatus, pr_reg) into the note data; that is part of the per-machine ELF formats userland knows about. In other cases, the core file note contains exactly the whole regset (n * `size`) and nothing else. The core file note is normally omitted when there is an `active` function and it returns zero.

struct user_regset_view

Name

struct user_regset_view — available regsets

Synopsis

```
struct user_regset_view {
  const char * name;
  const struct user_regset * regsets;
  unsigned int n;
  u32 e_flags;
  u16 e_machine;
  u8 ei_osabi;
};
```

Members

name

> Identifier, e.g. UTS_MACHINE string.

regsets

> Array of n regsets available in this view.

n

> Number of elements in $regsets$.

e_flags

> ELF header e_flags value written in core dumps.

e_machine

> ELF header $e\_machine$ EM_* value written in core dumps.

ei_osabi

> ELF header e_ident[EI_OSABI] value written in core dumps.

Description

A regset view is a collection of regsets (struct user_regset, above). This describes all the state of a thread that can be seen from a given architecture/ABI environment. More than one view might refer to the same struct user_regset, or more than one regset might refer to the same machine-specific state in the thread. For example, a 32-bit thread's state could be examined from the 32-bit view or from the 64-bit view. Either method reaches the same thread register state, doing appropriate widening or truncation.

task_user_regset_view

LINUX

Name

`task_user_regset_view` — Return the process's native regset view.

Synopsis

```
const struct user_regset_view * task_user_regset_view (struct
task_struct * tsk);
```

Arguments

tsk

> a thread of the process in question

Description

Return the struct user_regset_view that is native for the given process. For example, what it would access when it called `ptrace`. Throughout the life of the process, this only changes at exec.

copy_regset_to_user

Name

`copy_regset_to_user` — fetch a thread's user_regset data into user memory

Synopsis

```
int copy_regset_to_user (struct task_struct * target, const
struct user_regset_view * view, unsigned int setno, unsigned
int offset, unsigned int size, void __user * data);
```

Arguments

target

    thread to be examined

view

    struct user_regset_view describing user thread machine state

setno

    index in *view*->regsets

offset

    offset into the regset data, in bytes

size

    amount of data to copy, in bytes

data

    user-mode pointer to copy into

copy_regset_from_user

LINUX

Name

`copy_regset_from_user` — store into thread's user_regset data from user memory

Synopsis

```
int copy_regset_from_user (struct task_struct * target, const
struct user_regset_view * view, unsigned int setno, unsigned
int offset, unsigned int size, const void __user * data);
```

Arguments

target

> thread to be examined

view

> struct user_regset_view describing user thread machine state

setno

> index in *view*->regsets

offset

> offset into the regset data, in bytes

size

> amount of data to copy, in bytes

data

> user-mode pointer to copy from

3.2. `System Call Information`

This function is declared in `<linux/ptrace.h>`.

task_current_syscall

Name

`task_current_syscall` — Discover what a blocked task is doing.

Synopsis

```
int task_current_syscall (struct task_struct * target, long *
callno, unsigned long args[6], unsigned int maxargs, unsigned
long * sp, unsigned long * pc);
```

Arguments

`target`

> thread to examine

`callno`

> filled with system call number or -1

`args[6]`

> filled with `maxargs` system call arguments

`maxargs`

> number of elements in `args` to fill

`sp`

> filled with user stack pointer

```
pc
```

> filled with user PC

Description

If `target` is blocked in a system call, returns zero with `*callno` set to the the call's number and `args` filled in with its arguments. Registers not used for system call arguments may not be available and it is not kosher to use struct user_regset calls while the system call is still in progress. Note we may get this result if `target` has finished its system call but not yet returned to user mode, such as when it's stopped for signal handling or syscall exit tracing.

If `target` is blocked in the kernel during a fault or exception, returns zero with `*callno` set to -1 and does not fill in `args`. If so, it's now safe to examine `target` using struct user_regset `get` calls as long as we're sure `target` won't return to user mode.

Returns -EAGAIN if `target` does not remain blocked.

Returns -EINVAL if `maxargs` is too large (maximum is six).

3.3. `System Call Tracing`

The arch API for system call information is declared in `<asm/syscall.h>`. Each of these calls can be used only at system call entry tracing, or can be used only at system call exit and the subsequent safe points before returning to user mode. At system call entry tracing means either during a `report_syscall_entry` callback, or any time after that callback has returned UTRACE_STOP.

syscall_get_nr

Name

syscall_get_nr — find what system call a task is executing

Synopsis

```
int syscall_get_nr (struct task_struct * task, struct pt_regs
* regs);
```

Arguments

task

task of interest, must be blocked

regs

`task_pt_regs` of *task*

Description

If *task* is executing a system call or is at system call tracing about to attempt one, returns the system call number. If *task* is not executing a system call, i.e. it's blocked inside the kernel for a fault or signal, returns -1.

Note this returns int even on 64-bit machines. Only 32 bits of system call number can be meaningful. If the actual arch value is 64 bits, this truncates to 32 bits so 0xffffffff means -1.

It's only valid to call this when *task* is known to be blocked.

syscall_rollback

LINUX

Kernel Hackers ManualJuly 2010

Name

`syscall_rollback` — roll back registers after an aborted system call

Synopsis

```
void syscall_rollback (struct task_struct * task, struct
pt_regs * regs);
```

Arguments

task

> task of interest, must be in system call exit tracing

regs

> `task_pt_regs` of *task*

Description

It's only valid to call this when `task` is stopped for system call exit tracing (due to TIF_SYSCALL_TRACE or TIF_SYSCALL_AUDIT), after `tracehook_report_syscall_entry` returned nonzero to prevent the system call from taking place.

This rolls back the register state in `regs` so it's as if the system call instruction was a no-op. The registers containing the system call number and arguments are as they were before the system call instruction. This may not be the same as what the register state looked like at system call entry tracing.

syscall_get_error

LINUX

Kernel Hackers ManualJuly 2010

Name

`syscall_get_error` — check result of traced system call

Synopsis

```
long syscall_get_error (struct task_struct * task, struct
pt_regs * regs);
```

Arguments

task

   task of interest, must be blocked

regs

   `task_pt_regs` of *task*

Description

Returns 0 if the system call succeeded, or -ERRORCODE if it failed.

It's only valid to call this when *task* is stopped for tracing on exit from a system call, due to `TIF_SYSCALL_TRACE` or `TIF_SYSCALL_AUDIT`.

syscall_get_return_value

LINUX

Kernel Hackers ManualJuly 2010

Name

`syscall_get_return_value` — get the return value of a traced system call

Synopsis

```
long syscall_get_return_value (struct task_struct * task,
struct pt_regs * regs);
```

Arguments

task

> task of interest, must be blocked

regs

> `task_pt_regs` of *task*

Description

Returns the return value of the successful system call. This value is meaningless if `syscall_get_error` returned nonzero.

It's only valid to call this when *task* is stopped for tracing on exit from a system call, due to `TIF_SYSCALL_TRACE` or `TIF_SYSCALL_AUDIT`.

syscall_set_return_value

LINUX

Kernel Hackers ManualJuly 2010

Name

`syscall_set_return_value` — change the return value of a traced system call

Synopsis

```
void syscall_set_return_value (struct task_struct * task,
struct pt_regs * regs, int error, long val);
```

Arguments

task

task of interest, must be blocked

regs

`task_pt_regs` of *task*

error

negative error code, or zero to indicate success

val

user return value if *error* is zero

Description

This changes the results of the system call that user mode will see. If *error* is zero, the user sees a successful system call with a return value of *val*. If *error* is nonzero, it's a negated errno code; the user sees a failed system call with this errno code.

It's only valid to call this when *task* is stopped for tracing on exit from a system call, due to `TIF_SYSCALL_TRACE` or `TIF_SYSCALL_AUDIT`.

syscall_get_arguments

LINUX

Name

`syscall_get_arguments` — extract system call parameter values

Synopsis

```
void syscall_get_arguments (struct task_struct * task, struct
pt_regs * regs, unsigned int i, unsigned int n, unsigned long
* args);
```

Arguments

`task`

task of interest, must be blocked

`regs`

`task_pt_regs` of `task`

`i`

argument index [0,5]

`n`

number of arguments; n+i must be [1,6].

`args`

array filled with argument values

Description

Fetches `n` arguments to the system call starting with the `i`'th argument (from 0 through 5). Argument `i` is stored in `args`[0], and so on. An arch inline version is probably optimal when `i` and `n` are constants.

It's only valid to call this when `task` is stopped for tracing on entry to a system call, due to `TIF_SYSCALL_TRACE` or `TIF_SYSCALL_AUDIT`. It's invalid to call this with $i + n > 6$; we only support system calls taking up to 6 arguments.

syscall_set_arguments

Name

`syscall_set_arguments` — change system call parameter value

Synopsis

```
void syscall_set_arguments (struct task_struct * task, struct
pt_regs * regs, unsigned int i, unsigned int n, const unsigned
long * args);
```

Arguments

task

task of interest, must be in system call entry tracing

regs

`task_pt_regs` of `task`

i

argument index [0,5]

n

number of arguments; n+i must be [1,6].

`args`

>   array of argument values to store

Description

Changes n arguments to the system call starting with the i'th argument. Argument i gets value `args`[0], and so on. An arch inline version is probably optimal when i and n are constants.

It's only valid to call this when `task` is stopped for tracing on entry to a system call, due to `TIF_SYSCALL_TRACE` or `TIF_SYSCALL_AUDIT`. It's invalid to call this with $i + n > 6$; we only support system calls taking up to 6 arguments.

Chapter 4. Kernel Internals

This chapter covers the interface to the tracing infrastructure from the core of the kernel and the architecture-specific code. This is for maintainers of the kernel and arch code, and not relevant to using the tracing facilities described in preceding chapters.

4.1. Core Calls In

These calls are declared in `<linux/tracehook.h>`. The core kernel calls these functions at various important places.

tracehook_expect_breakpoints

Name

`tracehook_expect_breakpoints` — guess if task memory might be touched

Synopsis

int **tracehook_expect_breakpoints** (struct task_struct * *task*);

Arguments

task

current task, making a new mapping

Description

Return nonzero if `task` is expected to want breakpoint insertion in its memory at some point. A zero return is no guarantee it won't be done, but this is a hint that it's known to be likely.

May be called with `task`->mm->mmap_sem held for writing.

tracehook_report_syscall_entry

LINUX

Kernel Hackers ManualJuly 2010

Name

`tracehook_report_syscall_entry` — task is about to attempt a system call

Synopsis

`__must_check int` **`tracehook_report_syscall_entry`** `(struct pt_regs *` `regs`)`;`

Arguments

`regs`

　　user register state of current task

Description

This will be called if `TIF_SYSCALL_TRACE` has been set, when the current task has just entered the kernel for a system call. Full user register state is available here.

Changing the values in `regs` can affect the system call number and arguments to be tried. It is safe to block here, preventing the system call from beginning.

Returns zero normally, or nonzero if the calling arch code should abort the system call. That must prevent normal entry so no system call is made. If `task` ever returns to user mode after this, its register state is unspecified, but should be something harmless like an ENOSYS error return. It should preserve enough information so that `syscall_rollback` can work (see asm-generic/syscall.h).

Called without locks, just after entering kernel mode.

tracehook_report_syscall_exit

Name

`tracehook_report_syscall_exit` — task has just finished a system call

Synopsis

void **tracehook_report_syscall_exit** (struct pt_regs * *regs*, int *step*);

Arguments

regs

>   user register state of current task

step

>   nonzero if simulating single-step or block-step

Description

This will be called if `TIF_SYSCALL_TRACE` has been set, when the current task has just finished an attempted system call. Full user register state is available here. It is safe to block here, preventing signals from being processed.

If *step* is nonzero, this report is also in lieu of the normal trap that would follow the system call instruction because `user_enable_block_step` or `user_enable_single_step` was used. In this case, `TIF_SYSCALL_TRACE` might not be set.

Called without locks, just before checking for pending signals.

tracehook_unsafe_exec

Name

`tracehook_unsafe_exec` — check for exec declared unsafe due to tracing

Synopsis

```
int tracehook_unsafe_exec (struct task_struct * task);
```

Arguments

task

current task doing exec

Description

Return `LSM_UNSAFE_`* bits applied to an exec because of tracing.

`task`->cred_guard_mutex is held by the caller through the `do_execve`.

tracehook_tracer_task

LINUX

Name

`tracehook_tracer_task` — return the task that is tracing the given task

Synopsis

```
struct task_struct * tracehook_tracer_task (struct task_struct
* tsk);
```

Arguments

`tsk`

   task to consider

Description

Returns NULL if noone is tracing `task`, or the struct task_struct pointer to its tracer.

Must called under `rcu_read_lock`. The pointer returned might be kept live only by RCU. During exec, this may be called with `task_lock` held on `task`, still held from when `tracehook_unsafe_exec` was called.

tracehook_report_exec

Name

tracehook_report_exec — a successful exec was completed

Synopsis

```
void tracehook_report_exec (struct linux_binfmt * fmt, struct
linux_binprm * bprm, struct pt_regs * regs);
```

Arguments

fmt

>   struct linux_binfmt that performed the exec

bprm

>   struct linux_binprm containing exec details

regs

>   user-mode register state

Description

An exec just completed, we are shortly going to return to user mode. The freshly initialized register state can be seen and changed in *regs*. The name, file and other pointers in *bprm* are still on hand to be inspected, but will be freed as soon as this returns.

Called with no locks, but with some kernel resources held live and a reference on `fmt`->module.

tracehook_report_exit

Name

`tracehook_report_exit` — task has begun to exit

Synopsis

`void` **`tracehook_report_exit`** `(long * exit_code);`

Arguments

`exit_code`

> pointer to value destined for `current`->exit_code

Description

`exit_code` points to the value passed to `do_exit`, which tracing might change here. This is almost the first thing in `do_exit`, before freeing any resources or setting the `PF_EXITING` flag.

Called with no locks held.

tracehook_prepare_clone

Name

`tracehook_prepare_clone` — prepare for new child to be cloned

Synopsis

```
int tracehook_prepare_clone (unsigned clone_flags);
```

Arguments

clone_flags

> `CLONE_*` flags from clone/fork/vfork system call

Description

This is called before a new user task is to be cloned. Its return value will be passed to `tracehook_finish_clone`.

Called with no locks held.

tracehook_finish_clone

Name

`tracehook_finish_clone` — new child created and being attached

Synopsis

void **tracehook_finish_clone** (struct task_struct * *child*,
unsigned long *clone_flags*, int *trace*);

Arguments

child

> new child task

clone_flags

> CLONE_* flags from clone/fork/vfork system call

trace

> return value from `tracehook_prepare_clone`

Description

This is called immediately after adding *child* to its parent's children list. The *trace* value is that returned by `tracehook_prepare_clone`.

Called with current's siglock and write_lock_irq(tasklist_lock) held.

tracehook_report_clone

Name

`tracehook_report_clone` — in parent, new child is about to start running

Synopsis

```
void tracehook_report_clone (struct pt_regs * regs, unsigned
long clone_flags, pid_t pid, struct task_struct * child);
```

Arguments

regs

parent's user register state

clone_flags

flags from parent's system call

pid

new child's PID in the parent's namespace

child

new child task

Description

Called after a child is set up, but before it has been started running. This is not a good place to block, because the child has not started yet. Suspend the child here if desired, and then block in `tracehook_report_clone_complete`. This must prevent the child from self-reaping if `tracehook_report_clone_complete` uses the `child` pointer; otherwise it might have died and been released by the time `tracehook_report_clone_complete` is called.

Called with no locks held, but the child cannot run until this returns.

tracehook_report_clone_complete

Name

`tracehook_report_clone_complete` — new child is running

Synopsis

```
void tracehook_report_clone_complete (int trace, struct
pt_regs * regs, unsigned long clone_flags, pid_t pid, struct
task_struct * child);
```

Arguments

trace

> return value from `tracehook_prepare_clone`

regs

> parent's user register state

clone_flags

> flags from parent's system call

pid

> new child's PID in the parent's namespace

child

> child task, already running

Description

This is called just after the child has started running. This is just before the clone/fork syscall returns, or blocks for vfork child completion if `clone_flags` has the `CLONE_VFORK` bit set. The `child` pointer may be invalid if a self-reaping child died and `tracehook_report_clone` took no action to prevent it from self-reaping.

Called with no locks held.

tracehook_report_vfork_done

Name

`tracehook_report_vfork_done` — vfork parent's child has exited or exec'd

Synopsis

void **tracehook_report_vfork_done** (struct task_struct * *child*, pid_t *pid*);

Arguments

child

child task, already running

pid

new child's PID in the parent's namespace

Description

Called after a `CLONE_VFORK` parent has waited for the child to complete. The clone/vfork system call will return immediately after this. The `child` pointer may be invalid if a self-reaping child died and `tracehook_report_clone` took no action to prevent it from self-reaping.

Called with no locks held.

tracehook_prepare_release_task

Name

`tracehook_prepare_release_task` — task is being reaped, clean up tracing

Synopsis

```
void tracehook_prepare_release_task (struct task_struct *
task);
```

Arguments

`task`

task in `EXIT_DEAD` state

Description

This is called in `release_task` just before *task* gets finally reaped and freed. This would be the ideal place to remove and clean up any tracing-related state for *task*.

Called with no locks held.

tracehook_finish_release_task

Name

`tracehook_finish_release_task` — final tracing clean-up

Synopsis

```
void tracehook_finish_release_task (struct task_struct *
task);
```

Arguments

task

task in `EXIT_DEAD` state

Description

This is called in `release_task` when *task* is being in the middle of being reaped. After this, there must be no tracing entanglements.

Called with write_lock_irq(tasklist_lock) held.

tracehook_signal_handler

Name

`tracehook_signal_handler` — signal handler setup is complete

Synopsis

```
void tracehook_signal_handler (int sig, siginfo_t * info,
const struct k_sigaction * ka, struct pt_regs * regs, int
stepping);
```

Arguments

sig

    number of signal being delivered

info

    siginfo_t of signal being delivered

ka

    sigaction setting that chose the handler

regs

    user register state

stepping

    nonzero if debugger single-step or block-step in use

Description

Called by the arch code after a signal handler has been set up. Register and stack state reflects the user handler about to run. Signal mask changes have already been made.

Called without locks, shortly before returning to user mode (or handling more signals).

tracehook_consider_ignored_signal

LINUX

Kernel Hackers ManualJuly 2010

Name

`tracehook_consider_ignored_signal` — suppress short-circuit of ignored signal

Synopsis

```
int tracehook_consider_ignored_signal (struct task_struct *
task, int sig);
```

Arguments

task

> task receiving the signal

sig

> signal number being sent

Description

Return zero iff tracing doesn't care to examine this ignored signal, so it can short-circuit normal delivery and never even get queued.

Called with *task*->sighand->siglock held.

tracehook_consider_fatal_signal

Name

tracehook_consider_fatal_signal — suppress special handling of fatal signal

Synopsis

int **tracehook_consider_fatal_signal** (struct task_struct * *task*, int *sig*);

Arguments

task

    task receiving the signal

sig

    signal number being sent

Description

Return nonzero to prevent special handling of this termination signal. Normally handler for signal is `SIG_DFL`. It can be `SIG_IGN` if *sig* is ignored, in which case `force_sig` is about to reset it to `SIG_DFL`. When this returns zero, this signal might cause a quick termination that does not give the debugger a chance to intercept the signal.

Called with or without *task*->sighand->siglock held.

tracehook_force_sigpending

Name

`tracehook_force_sigpending` — let tracing force signal_pending(current) on

Synopsis

```
int tracehook_force_sigpending ( void);
```

Arguments

void

no arguments

Description

Called when recomputing our `signal_pending` flag. Return nonzero to force the `signal_pending` flag on, so that `tracehook_get_signal` will be called before the next return to user mode.

Called with *current*->sighand->siglock held.

tracehook_get_signal

Name

`tracehook_get_signal` — deliver synthetic signal to traced task

Synopsis

```
int tracehook_get_signal (struct task_struct * task, struct
pt_regs * regs, siginfo_t * info, struct k_sigaction *
return_ka);
```

Arguments

task

   current

regs

   task_pt_regs(*current*)

info

   details of synthetic signal

```
return_ka
```

sigaction for synthetic signal

Description

Return zero to check for a real pending signal normally. Return -1 after releasing the siglock to repeat the check. Return a signal number to induce an artifical signal delivery, setting *`info` and *`return_ka` to specify its details and behavior.

The `return_ka`->sa_handler value controls the disposition of the signal, no matter the signal number. For `SIG_DFL`, the return value is a representative signal to indicate the behavior (e.g. `SIGTERM` for death, `SIGQUIT` for core dump, `SIGSTOP` for job control stop, `SIGTSTP` for stop unless in an orphaned pgrp), but the signal number reported will be `info`->si_signo instead.

Called with `task`->sighand->siglock held, before dequeuing pending signals.

tracehook_notify_jctl

Name

`tracehook_notify_jctl` — report about job control stop/continue

Synopsis

```
int tracehook_notify_jctl (int notify, int why);
```

Arguments

notify

> zero, `CLD_STOPPED` or `CLD_CONTINUED`

why

> `CLD_STOPPED` or `CLD_CONTINUED`

Description

This is called when we might call `do_notify_parent_cldstop`.

notify is zero if we would not ordinarily send a `SIGCHLD`, or is the `CLD_STOPPED` or `CLD_CONTINUED` .si_code for `SIGCHLD`.

why is `CLD_STOPPED` when about to stop for job control; we are already in `TASK_STOPPED` state, about to call `schedule`. It might also be that we have just exited (check `PF_EXITING`), but need to report that a group-wide stop is complete.

why is `CLD_CONTINUED` when waking up after job control stop and ready to make a delayed *notify* report.

Return the `CLD_*` value for `SIGCHLD`, or zero to generate no signal.

Called with the siglock held.

tracehook_finish_jctl

Name

`tracehook_finish_jctl` — report about return from job control stop

Synopsis

```
void tracehook_finish_jctl ( void);
```

Arguments

void

> no arguments

Description

This is called by `do_signal_stop` after wakeup.

tracehook_notify_death

Name

`tracehook_notify_death` — task is dead, ready to notify parent

Synopsis

```
int tracehook_notify_death (struct task_struct * task, void **
death_cookie, int group_dead);
```

Arguments

task

> *current* task now exiting

death_cookie

> value to pass to `tracehook_report_death`

group_dead

> nonzero if this was the last thread in the group to die

Description

A return value >= 0 means call `do_notify_parent` with that signal number. Negative return value can be `DEATH_REAP` to self-reap right now, or `DEATH_DELAYED_GROUP_LEADER` to a zombie without notifying our parent. Note that a return value of 0 means a `do_notify_parent` call that sends no signal, but still wakes up a parent blocked in wait*().

Called with write_lock_irq(tasklist_lock) held.

tracehook_report_death

Name

`tracehook_report_death` — task is dead and ready to be reaped

Synopsis

```
void tracehook_report_death (struct task_struct * task, int
signal, void * death_cookie, int group_dead);
```

Arguments

`task`

> `current` task now exiting

`signal`

> return value from `tracheook_notify_death`

`death_cookie`

> value passed back from `tracehook_notify_death`

`group_dead`

> nonzero if this was the last thread in the group to die

Description

Thread has just become a zombie or is about to self-reap. If positive, `signal` is the signal number just sent to the parent (usually `SIGCHLD`). If `signal` is `DEATH_REAP`, this thread will self-reap. If `signal` is `DEATH_DELAYED_GROUP_LEADER`, this is a `delayed_group_leader` zombie. The `death_cookie` was passed back by `tracehook_notify_death`.

If normal reaping is not inhibited, `task`->exit_state might be changing in parallel.

Called without locks.

set_notify_resume

LINUX

Kernel Hackers ManualJuly 2010

Name

set_notify_resume — cause `tracehook_notify_resume` to be called

Synopsis

```
void set_notify_resume (struct task_struct * task);
```

Arguments

`task`

>   task that will call `tracehook_notify_resume`

Description

Calling this arranges that *task* will call `tracehook_notify_resume` before returning to user mode. If it's already running in user mode, it will enter the kernel and call `tracehook_notify_resume` soon. If it's blocked, it will not be woken.

tracehook_notify_resume

Name

`tracehook_notify_resume` — report when about to return to user mode

Synopsis

```
void tracehook_notify_resume (struct pt_regs * regs);
```

Arguments

regs

> user-mode registers of `current` task

Description

This is called when `TIF_NOTIFY_RESUME` has been set. Now we are about to return to user mode, and the user state in `regs` can be inspected or adjusted. The caller in arch code has cleared `TIF_NOTIFY_RESUME` before the call. If the flag gets set again asynchronously, this will be called again before we return to user mode.

Called without locks. However, on some machines this may be called with interrupts disabled.

4.2. Architecture Calls Out

An arch that has done all these things sets `CONFIG_HAVE_ARCH_TRACEHOOK`. This is required to enable the utrace code.

4.2.1. `<asm/ptrace.h>`

An arch defines these in `<asm/ptrace.h>` if it supports hardware single-step or block-step features.

arch_has_single_step

Name

`arch_has_single_step` — does this CPU support user-mode single-step?

Synopsis

arch_has_single_step `(void);`

Arguments

None

Description

If this is defined, then there must be function declarations or inlines for `user_enable_single_step` and `user_disable_single_step`. `arch_has_single_step` should evaluate to nonzero iff the machine supports instruction single-step for user mode. It can be a constant or it can test a CPU feature bit.

arch_has_block_step

Name

`arch_has_block_step` — does this CPU support user-mode block-step?

Synopsis

arch_has_block_step `(void);`

Arguments

None

Description

If this is defined, then there must be a function declaration or inline for `user_enable_block_step`, and `arch_has_single_step` must be defined too. `arch_has_block_step` should evaluate to nonzero iff the machine supports step-until-branch for user mode. It can be a constant or it can test a CPU feature bit.

user_enable_single_step

LINUX

Kernel Hackers ManualJuly 2010

Name

`user_enable_single_step` — single-step in user-mode task

Synopsis

```
void user_enable_single_step (struct task_struct * task);
```

Arguments

`task`

> either current or a task stopped in `TASK_TRACED`

Description

This can only be called when `arch_has_single_step` has returned nonzero. Set `task` so that when it returns to user mode, it will trap after the next single instruction executes. If `arch_has_block_step` is defined, this must clear the effects of `user_enable_block_step` too.

user_enable_block_step

Name

user_enable_block_step — step until branch in user-mode task

Synopsis

void **user_enable_block_step** (struct task_struct * *task*);

Arguments

task

either current or a task stopped in `TASK_TRACED`

Description

This can only be called when `arch_has_block_step` has returned nonzero, and will never be called when single-instruction stepping is being used. Set `task` so that when it returns to user mode, it will trap after the next branch or trap taken.

user_disable_single_step

Name

`user_disable_single_step` — cancel user-mode single-step

Synopsis

`void` **`user_disable_single_step`** `(struct task_struct * task);`

Arguments

`task`

> either current or a task stopped in `TASK_TRACED`

Description

Clear `task` of the effects of `user_enable_single_step` and `user_enable_block_step`. This can be called whether or not either of those was ever called on `task`, and even if `arch_has_single_step` returned zero.

4.2.2. `<asm/syscall.h>`

An arch provides `<asm/syscall.h>` that defines these as inlines, or declares them as exported functions. These interfaces are described in Section 3.3.

4.2.3. `<linux/tracehook.h>`

An arch must define `TIF_NOTIFY_RESUME` and `TIF_SYSCALL_TRACE` in its
`<asm/thread_info.h>`. The arch code must call the following functions, all
declared in `<linux/tracehook.h>` and described in Section 4.1:

- `tracehook_notify_resume`

- `tracehook_report_syscall_entry`

- `tracehook_report_syscall_exit`

- `tracehook_signal_handler`