

TASK_KILLABLE: New process state in Linux

This new sleeping state echoes TASK_UNINTERRUPTIBLE with the ability to respond to fatal signals

Skill Level: Introductory

[Avinesh Kumar \(avinesh.kumar@in.ibm.com\)](mailto:avinesh.kumar@in.ibm.com)

System Software Engineer
IBM

30 Sep 2008

Linux® kernel 2.6.25 introduced a new process state for putting processes to sleep called TASK_KILLABLE, which offers an alternative to the efficient but potentially unkillable TASK_UNINTERRUPTIBLE and the easy-to-awaken but safer TASK_INTERRUPTIBLE. TASK_KILLABLE is the outcome of an issue raised in 2002 about the OpenAFS file system driver waiting for an event interruptibly after blocking all signals. This new sleeping state echoes TASK_UNINTERRUPTIBLE with the ability to respond to fatal signals. In this article, the author sheds light on this area and, using examples from 2.6.26 and an earlier version, 2.6.18, discusses the related changes to the Linux kernel and the new APIs that resulted from these changes.

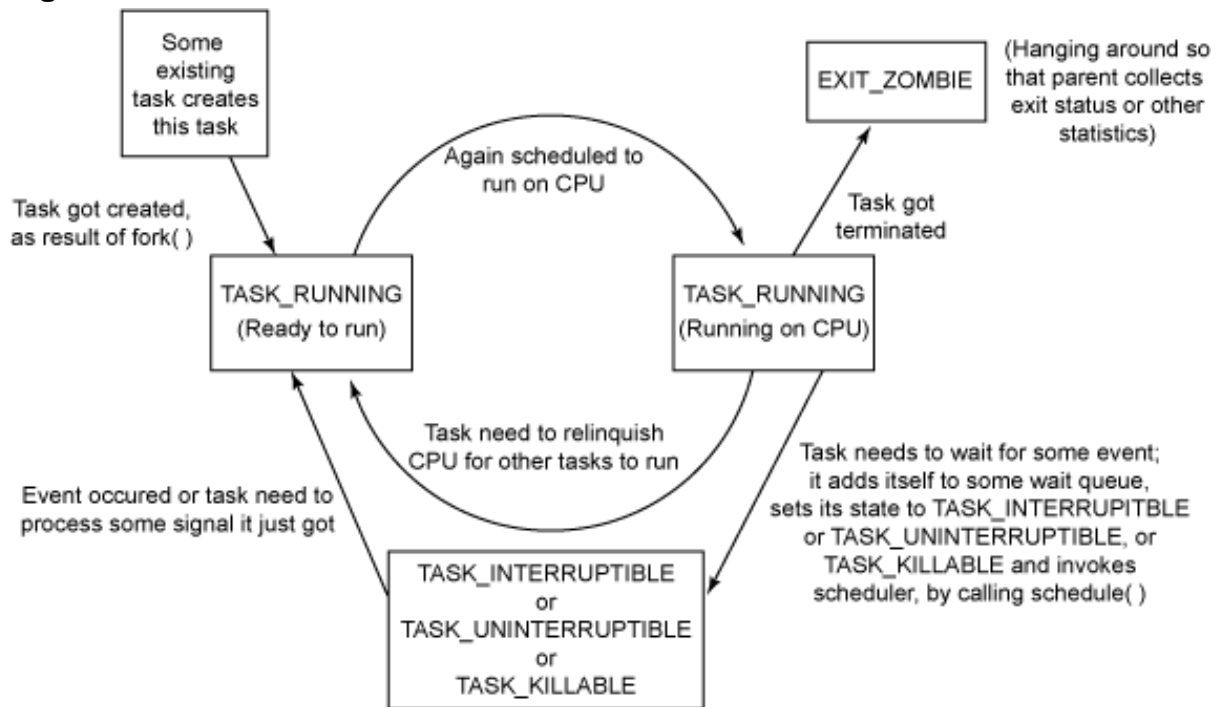
Like files, processes are fundamental to any UNIX® operating system. Processes are live entities executing the instructions of an executable file. Apart from executing its instructions, a process might be engaged in managing open files, processor context, address space, and data related to the program, among other things. The Linux kernel keeps complete information about a process in a *process descriptor* defined as `struct task_struct`. You can see the various fields of `struct task_struct` in the Linux kernel source file `include/linux/sched.h`.

About process states

During its lifetime, a process may go through a set of mutually exclusive states. The kernel keeps the state information of a process in the *state* field of `struct`

task_struct. Figure 1 shows the transition between the process states.

Figure 1. Process state transition



Let's review the various process states:

- **TASK_RUNNING**: The process is either running on CPU or waiting in a run queue to get scheduled.
- **TASK_INTERRUPTIBLE**: The process is sleeping, waiting for some event to occur. The process is open to be interrupted by signals. Once signalled or awoken by explicit wake-up call, the process transitions to **TASK_RUNNING**.
- **TASK_UNINTERRUPTIBLE**: The process state is similar to **TASK_INTERRUPTIBLE** except that in this state it does not process signals. It may not be desirable even to interrupt the process while in this state since it may be in the middle of completing some important task. When the event occurs that it is waiting for, the process is awoken by the explicit wake-up call.
- **TASK_STOPPED**: The process execution is stopped, it's not running, and not able to run. On receipt of signals like **SIGSTOP**, **SIGTSTP**, and so on, the process arrives at this state. The process would be runnable again on receipt of signal **SIGCONT**.
- **TASK_TRACED**: A process arrives at this state while it is being monitored by other processes such as debuggers.

- `EXIT_ZOMBIE`: The process has terminated. It is lingering around just for its parent to collect some statistical information about it.
- `EXIT_DEAD`: The final state (just like it sounds). The process reaches this state when it is being removed from the system since its parent has just collected all statistical information by issuing the `wait4()` or `waitpid()` system call.

For detailed information on process state transitions, see *The Design of the UNIX Operating System* listed in the [Resources](#) section.

As noted, the process states `TASK_UNINTERRUPTIBLE` and `TASK_INTERRUPTIBLE` are sleeping states. Now let's look at the mechanism provided by the kernel for putting a process to sleep.

Kernel napping

The Linux kernel provides two ways to put a process to sleep.

The normal way to put a process to sleep is to set the process's state to either `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` and call the scheduler's function `schedule()`. This results in the process getting moved off from the CPU run queue. If the process is sleeping in interruptible mode (by setting its state to `TASK_INTERRUPTIBLE`), it can be awakened either by an explicit wake-up call (`wakeup_process()`) or by signals needing processing.

However, if the process is sleeping in uninterruptible mode (by setting its state to `TASK_UNINTERRUPTIBLE`), it can only be awakened by an explicit wake-up call. It is advised to put the processes into interruptible sleep mode rather than uninterruptible sleep mode unless you really, really need to (such during device I/O when processing signals is difficult).

When a task sleeping interruptibly receives a signal, it will have to process the signal (unless its already masked!), leaving whatever it was doing (cleanup code required here) and return `-EINTR` back to the user space. Again, the responsibility of checking this return code and taking appropriate action lies with the programmer. So a lazy programmer might prefer putting the process into uninterruptible sleeping mode because signals do not wake up such tasks. But be cautious of cases where the wakeup call on the process sleeping uninterruptibly could not happen for some reason, making the process unkillable, which will eventually cause frustration since the only way out is a system reboot. On one hand, you need to take care of some details, because doing otherwise will introduce bugs both on the kernel and on the user side. On the other hand, you may get deadly immortals (blocked and unkillable processes).

Now we have a new way of sleeping in the kernel!

New sleeping state: TASK_KILLABLE

The Linux Kernel version 2.6.25 introduces a new process sleeping state, TASK_KILLABLE: If a process is sleeping killably in this new state, it works like TASK_UNINTERRUPTIBLE with the bonus that it can respond to fatal signals. Take a look at Listing 1 for a comparison of process states (as defined in *include/linux/sched.h*) from kernel 2.6.18 to 2.6.26:

Listing 1. Comparison of process states between 2.6.18 and 2.6.26

Linux Kernel 2.6.18		Linux Kernel 2.6.26	
=====		=====	
#define TASK_RUNNING	0	#define TASK_RUNNING	0
#define TASK_INTERRUPTIBLE	1	#define TASK_INTERRUPTIBLE	1
#define TASK_UNINTERRUPTIBLE	2	#define TASK_UNINTERRUPTIBLE	2
#define TASK_STOPPED	4	#define __TASK_STOPPED	4
#define TASK_TRACED	8	#define __TASK_TRACED	8
/* in tsk->exit_state */		/* in tsk->exit_state */	
#define EXIT_ZOMBIE	16	#define EXIT_ZOMBIE	16
#define EXIT_DEAD	32	#define EXIT_DEAD	32
/* in tsk->state again */		/* in tsk->state again */	
#define TASK_NONINTERACTIVE	64	#define TASK_DEAD	64
		#define TASK_WAKEKILL	128

Notice that the states TASK_INTERRUPTIBLE and TASK_UNINTERRUPTIBLE are not altered. TASK_WAKEKILL is designed to wake the process on receipt of fatal signals.

Listing 2 shows how the states TASK_STOPPED and TASK_TRACED are altered (along with the definition of TASK_KILLABLE):

Listing 2. New state definitions in kernel 2.6.26

```
#define TASK_KILLABLE      (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED      (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED      (TASK_WAKEKILL | __TASK_TRACED)
```

In other words, TASK_UNINTERRUPTIBLE + TASK_WAKEKILL = TASK_KILLABLE.

New kernel APIs using TASK_KILLABLE

A few words about completion

A completion mechanism is a good bet when you want to put a task to sleep but then wake it up when some event completes. It provides a simple way to synchronize without race conditions. The routine `wait_for_completion(struct completion *comp)`

makes the calling task sleep uninterruptibly unless the completion has happened. It requires that it is awakened by the `complete(struct completion *comp)` or `complete_all(struct completion *comp)` function.

Apart from `wait_for_completion_killable()`, other routines for waiting are:

- `wait_for_completion_timeout()`
- `wait_for_completion_interruptible()`
- `wait_for_completion_interruptible_timeout()`

See the definition of completion structure in `include/linux/completion.h`.

Let's look at some of the new functions in this new state.

- **`int wait_event_killable(wait_queue_t queue, condition);`**
This function is defined in `include/linux/wait.h`; it puts the calling process to sleep killably in `queue` until the `condition` evaluates to *true*.
- **`long schedule_timeout_killable(signed long timeout);`**
This is defined in `kernel/timer.c`; this routine basically sets the current task's state to `TASK_KILLABLE` and calls `schedule_timeout()`, which makes the calling task sleep for `timeout` number of *jiffies*. (In UNIX systems, a *jiffy* is basically the time between two consecutive clock ticks.)
- **`int wait_for_completion_killable(struct completion *comp);`**
Defined in `kernel/sched.c`, this routine is used to wait killably for the completion of an event. This function calls `schedule_timeout()` for `MAX_SCHEDULE_TIMEOUT` (defined to be equal to `LONG_MAX`) jiffies if there are no fatal signals pending.
- **`int mutex_lock_killable(struct mutex *lock);`**
Defined in `kernel/mutex.c`, this routine is used to acquire mutex lock. However, if the lock is not available and the task is waiting to get the lock, and in the meantime it gets a fatal signal, the task would be removed from the list of waiters waiting for the mutex lock to process the signal.
- **`int down_killable(struct semaphore *sem);`**
Defined in `kernel/semaphore.c`, it is used to acquire the semaphore `sem`. If the semaphore is not available, it's put to sleep; if a fatal signal is delivered to it, it would be removed from the waiters' list and would have to respond to the signal. The other two methods of acquiring a semaphore are by using the routines `down()` or `down_interruptible()`. The

function `down()` is deprecated now; you should use either `down_killable()` or `down_interruptible()`.

Changes in NFS client code

Some changes in NFS client code use this new process state. Listing 3 shows the difference the `nfs_wait_event` macro between Linux kernels 2.6.18 and 2.6.26.

Listing 3. Changes in `nfs_wait_event` due to `TASK_KILLABLE`

Linux Kernel 2.6.18	Linux Kernel 2.6.26
=====	=====
#define nfs_wait_event(clnt, wq, condition)	#define nfs_wait_event(clnt, wq,
condition)	condition)
{	{
int __retval = 0;	int __retval =
	wait_event_killable(wq,
condition);	__retval;
if (clnt->cl_intr) {	})
sigset_t oldmask;	
rpc_clnt_sigmask(clnt, &oldmask);	
__retval =	
wait_event_interruptible(wq, condition);	
rpc_clnt_sigunmask(clnt, &oldmask);	
} else	
wait_event(wq, condition);	
__retval;	
}	

Listing 4 shows the definition of function `nfs_direct_wait()` in Linux Kernels 2.6.18 vs. 2.6.26.

Listing 4. Changes to `nfs_direct_wait()` due to `TASK_KILLABLE`

```

Linux Kernel 2.6.18
=====
static ssize_t nfs_direct_wait(struct nfs_direct_req *dreq)
{
    ssize_t result = -EIOCBQUEUED;

    /* Async requests don't wait here */
    if (dreq->iocb)
        goto out;

    result = wait_for_completion_interruptible(&dreq->completion);

    if (!result)
        result = dreq->error;
    if (!result)
        result = dreq->count;

out:
    kref_put(&dreq->kref, nfs_direct_req_release);
    return (ssize_t) result;
}

```

```
Linux Kernel 2.6.26
=====
static ssize_t nfs_direct_wait(struct nfs_direct_req *dreq)
{
    ssize_t result = -EIOCBQUEUED;
    /* Async requests don't wait here */
    if (dreq->iocb)
        goto out;

    result = wait_for_completion_killable(&dreq->completion);
    if (!result)
        result = dreq->error;
    if (!result)
        result = dreq->count;
out:
    return (ssize_t) result;
}
```

To see more of the changes in the NFS client to take advantage of the new functionality, see the Linux Kernel Mailing List entry in [Resources](#).

The earlier NFS mount option `intr` was a way out of interrupting NFS client processes waiting for some event, but it allowed all interruptions, not just one by deadly signals (like `TASK_KILLABLE`).

Summary

Even though this feature is generally an improvement over the existing options—after all, it is another way to keep from getting stuck with dead processes—it will take some time to enter general use. Just remember, unless it is *really important* to prohibit every kind of interruption other than the explicit wake-up call (with the traditional `TASK_UNINTERRUPTIBLE`), go with the new `TASK_KILLABLE`.

Resources

Learn

- The [TASK_KILLABLE process state](#) arose from an issue raised by David Howells in 2002; he talked about how the OpenAFS file system driver blocks all signals interruptibly while waiting for an event, whereas they should actually be waiting in the TASK_UNINTERRUPTIBLE state.
- Jonathan Corbet's discussion of [TASK_KILLABLE](#) (LWN.net, July 2008) is a helpful introduction.
- "Kernel Korner: Sleeping in the Kernel" (Linux Journal, July 2005) explains the use of sleep in the Linux kernel.
- In *The Design of the UNIX Operating System* (Prentice Hall, 1986, Maurice J. Bach), Chapter 6 has excellent detailed information on process state transitions.
- The thread "[NFS Killable tasks request comments on patch](#)" in the [Linux Kernel Mailing List](#) demonstrates more of NFS client changes designed to take advantage of the new 2.6.26 TASK_KILLABLE function.
- Read "[Anatomy of the Linux kernel](#)" (developerWorks, June 2007) for an overview of how and why the kernel is structured the way it is.
- In the [developerWorks Linux zone](#), find more resources for Linux developers (including developers who are [new to Linux programming and system administration](#)), and scan our [most popular articles and tutorials](#).
- See all [Linux tips](#) and [Linux tutorials](#) on developerWorks.
- Stay current with [developerWorks technical events and Webcasts](#).

Get products and technologies

- [Order the SEK for Linux](#), a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With [IBM trial software](#), available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the [developerWorks community](#) through blogs, forums, podcasts, and spaces.

About the author

Avinesh Kumar

Avinesh Kumar works as a System Software Engineer for the Andrew File System Team at the IBM Software Labs in Pune, India. He works with kernel- and user-level debugging of dumps and crashes, as well as reported bugs on the Linux, AIX, and Solaris platforms. Avinesh has an MCA from the Department of Computer Science at the University of Pune. He is a Linux enthusiast who spends his spare time exploring the Linux kernel with his Hardy Heron (Ubuntu 8.04) box.

Trademarks

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Tivoli, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. See the current list of [IBM trademarks](#).

Linux is a trademark of Linus Torvalds in the United States, other countries, or both. UNIX is a registered trademark of The Open Group in the United States and other countries.