进程管理：详解基于Linux有几种进程状态

玩转Linux内核
官方社区最新信息搜集、文章推送、教程学习、技巧分享等~

进程生命周期

在Linux内核里，无论是进程还是线程，统一使用 task_struct{} 结构体来表示，也就是统一抽象为任务（task）。task_struct{} 定义在 include/linux/sched.h 文件中，十分复杂，这里简单了解下。

```
// include/linux/sched.h
// ... 省略

struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup (see current_thread_info()), this
         * must be the first element of task_struct.
         */
        struct thread_info              thread_info;
#endif
        /* -1 unrunnable, 0 runnable, >0 stopped: */
        volatile long                   state;
        int                             exit_state;
        int                             exit_code;
        int                             exit_signal;

        /*
         * This begins the randomizable portion of task_struct. Only
         * scheduling-critical items should be added above here.
         */
        randomized_struct_fields_start

        void                            *stack;
        refcount_t                      usage;
        /* Per task flags (PF_*), defined further below: */
        unsigned int                    flags;
        unsigned int                    ptrace;

#ifdef CONFIG_SMP
        int                             on_cpu;
        struct __call_single_node       wake_entry;
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /* Current CPU: */
        unsigned int                    cpu;
#endif
        unsigned int                    wakee_flips;
        unsigned long                   wakee_flip_decay_ts;
```

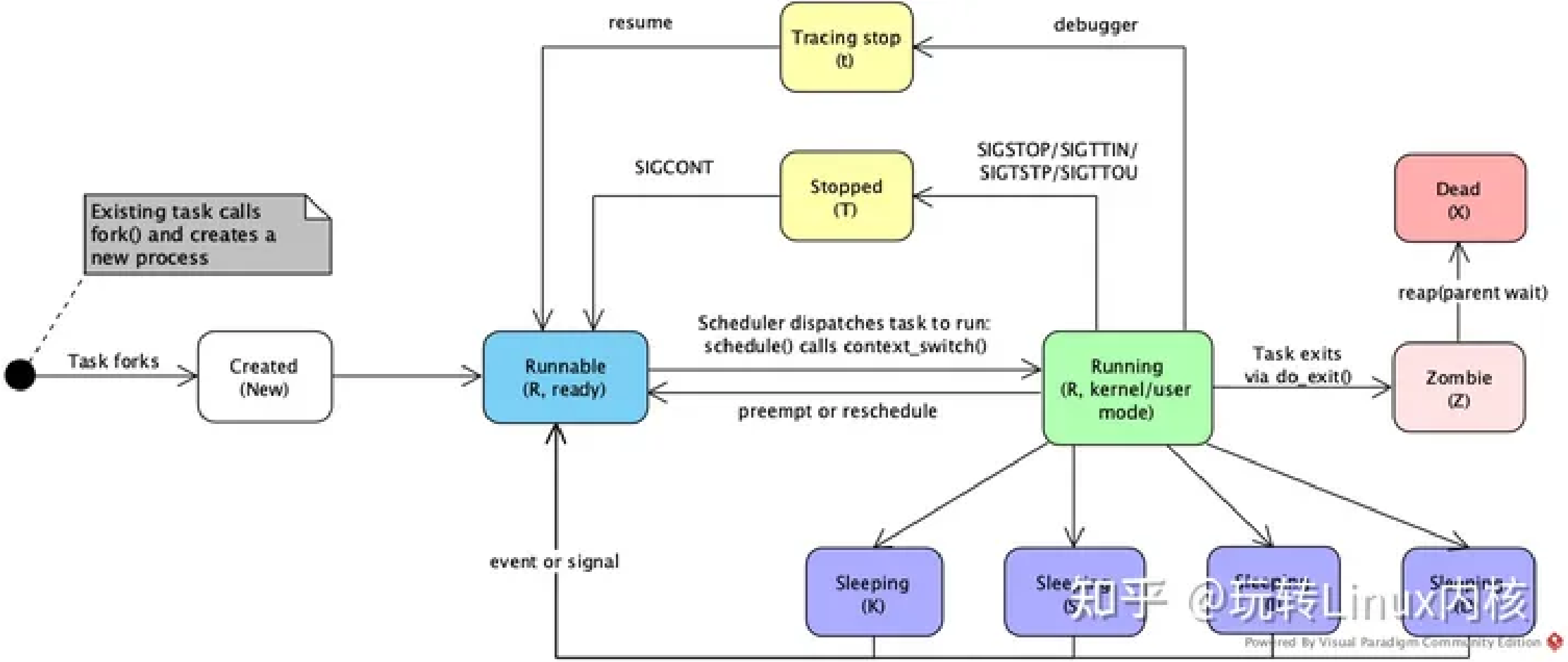
```
        struct task_struct              *last_wakee;
    // ...省略
    struct sched_info           sched_info;
        struct list_head                tasks;   // 链表，将所有task_struct串起来

        pid_t                           pid;    // process id，指的是线程id
        pid_t                           tgid;   // thread group ID，指的是进程的主线程id
    struct task_struct          *group_leader;  // 指向的是进程的主线程

        /* Signal handlers: */
        struct signal_struct            *signal;
        struct sighand_struct __rcu             *sighand;
        sigset_t                        blocked;
        sigset_t                        real_blocked;
        /* Restored if set_restore_sigmask() was used: */
        sigset_t                        saved_sigmask;
        struct sigpending               pending;
        unsigned long                   sas_ss_sp;
        size_t                          sas_ss_size;
        unsigned int                    sas_ss_flags;

    // ... 省略
}
```

查阅相关资料后，对Linux中进程的生命周期总结如下：



从图上可以看出，进程的睡眠状态是最多的，那进程一般在何时进入睡眠状态呢？答案是I/O操作时，因为I/O操作的速度与CPU运行速度相比，相差太大，所以此时进程会释放CPU，进入睡眠状态。

FLag	State	Name	Description
R	TASK_RUNNING	Runnable	在调度队列中等待分配CPU时间片
R	TASK_RUNNING	Running	获得CPU时间片，正在运行
Z	EXIT_ZOMBIE	Zombie	进程要结束，先进入该状态，相关资源（如内存等）已释放，但进程描述符，PID等资源还未释放，需要父进程使用wait()等系统调用来回收
X	EXIT_DEAD	Dead	父进程回收完子进程资源，进程结束。该状态无法通过ps等命令观察到
T	TASK_STOPPED	Stopped	中止状态，进程收到SIGSTOP/SIGTTIN/SIGTSTP/SIGTTOU等信号之后会进入该状态
t	TASK_TRACED	Tracing stop	表示进程被debugger等进程监视，进程执行被调试程序所停止。当一个进程被另外的进程所监视，每一个信号都会让进程进入该状态
			不可中断的睡眠状态，即深度睡眠，忽略

D	TASK_UNINTERRUPTIBLE	Disk Sleep	任何信号，只能死等I/O操作完成。一旦I/O操作因为特殊原因无法完成，则只能重启机器，所以是一个很危险的状态
K	TASK_KILLABLE	KILLABLE	基于TASK_UNINTERRUPTIBLE状态的进程无法被唤醒，所以有了TASK_KILLABLE，其原理与TASK_UNINTERRUPTIBLE相似，但它可以响应致命信号
S	TASK_INTERRUPTIBLE	Interruptible Sleep	可中断睡眠状态，即浅睡眠。虽然进程因等待I/O操作完成而进入睡眠，但此时来了一个信号，进程还是会被唤醒，用户可以按需定制信号处理函数
I	TASK_REPORT_IDLE	Idle	空闲状态，用在不可中断睡眠的内核线程上。也是D状态的一个子集，因为某些内核线程，它们实际上没有任何负载，所以使用I状态进行区分。在统计平均负载时，D状态的进程需要统计，而I状态不会，与K状态类似，它也响应致命信号

进程状态相关的定义同样在 include/linux/sched.h 文件的开头部分，以 #define TASK_KILLABLE (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE) 为例，TASK_WAKEKILL表示用于在接收到致命信号时唤醒进程，将它与 TASK_UNINTERRUPTIBLE 按位或，就得到了 TASK_KILLABLE。代码注释中提到了 fs/proc/array.c，所以也将其代码贴出，作为补充。

```
// include/linux/sched.h
// 完整文件github地址 https://github.com/torvalds/linux/blob/master/include/linux/sched.h#L86
// 省略...
/*
 * Task state bitmask. NOTE! These bits are also
 * encoded in fs/proc/array.c: get_task_state().
 *
 * We have two separate sets of flags: task->state
 * is about runnability, while task->exit_state are
 * about the task exiting. Confusing, but this way
 * modifying one set can't modify the other one by
 * mistake.
 */

/* Used in tsk->state: */
#define TASK_RUNNING                    0x0000
#define TASK_INTERRUPTIBLE              0x0001
#define TASK_UNINTERRUPTIBLE            0x0002
#define __TASK_STOPPED                  0x0004
#define __TASK_TRACED                   0x0008
/* Used in tsk->exit_state: */
#define EXIT_DEAD                       0x0010
#define EXIT_ZOMBIE                     0x0020
#define EXIT_TRACE                      (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->state again: */
#define TASK_PARKED                     0x0040
#define TASK_DEAD                       0x0080
#define TASK_WAKEKILL                   0x0100
#define TASK_WAKING                     0x0200
#define TASK_NOLOAD                     0x0400
#define TASK_NEW                        0x0800
#define TASK_STATE_MAX                  0x1000

/* Convenience macros for the sake of set_current_state: */
#define TASK_KILLABLE                   (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED                    (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED                     (TASK_WAKEKILL | __TASK_TRACED)

#define TASK_IDLE                       (TASK_UNINTERRUPTIBLE | TASK_NOLOAD)

/* Convenience macros for the sake of wake_up(): */
#define TASK_NORMAL                     (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)

/* get_task_state(): */
#define TASK_REPORT                     (TASK_RUNNING | TASK_INTERRUPTIBLE | \
                                         TASK_UNINTERRUPTIBLE | __TASK_STOPPED | \
                                         __TASK_TRACED | EXIT_DEAD | EXIT_ZOMBIE | \
                                         TASK_PARKED)

#define task_is_traced(task)            ((task->state & __TASK_TRACED) != 0)
```

```
#define task_is_stopped(task)           ((task->state & __TASK_STOPPED) != 0)

#define task_is_stopped_or_traced(task) ((task->state & (__TASK_STOPPED | __TASK_TRACED)) != 0)

// ... 省略
// fs/proc/array.c
// ... 省略
// 完整文件github地址 https://github.com/torvalds/linux/blob/master/fs/proc/array.c

/*
 * The task state array is a strange "bitmap" of
 * reasons to sleep. Thus "running" is zero, and
 * you can test for combinations of others with
 * simple bit tests.
 */
static const char * const task_state_array[] = {

        /* states in TASK_REPORT: */
        "R (running)",          /* 0x00 */
        "S (sleeping)",         /* 0x01 */
        "D (disk sleep)",       /* 0x02 */
        "T (stopped)",          /* 0x04 */
        "t (tracing stop)",     /* 0x08 */
        "X (dead)",             /* 0x10 */
        "Z (zombie)",           /* 0x20 */
        "P (parked)",           /* 0x40 */

        /* states beyond TASK_REPORT: */
        "I (idle)",             /* 0x80 */
};

static inline const char *get_task_state(struct task_struct *tsk)
{
        BUILD_BUG_ON(1 + ilog2(TASK_REPORT_MAX) != ARRAY_SIZE(task_state_array));
        return task_state_array[task_state_index(tsk)];
}

// ... 省略
```

在单核的CPU上，同一时刻只有一个task会被调度，所以即使看到了 R 状态，也不代表进程就被分配到了CPU时间片。但了解了进程状态之后，我们再通过 top, ps aux 等命令查看进程，**分析问题起来效率就更高了:**

```
top - 17:24:07 up 10:20,  1 user,  load average: 0.15, 0.08, 0.02
Tasks: 216 total,   1 running, 215 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.4 us,  0.3 sy,  0.0 ni, 99.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   5945.2 total,   2655.4 free,   1580.8 used,   1709.1 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   4084.6 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1914 demonlee  20   0 5252100 395940 132816 S   1.0   6.5   1:33.09 gnome-shell
    824 root      20   0 2495652  89512  48936 S   0.7   1.5   0:05.59 dockerd
```

```
 1687 demonlee  20   0 1153156  81436  50128 S   0.3   1.3   0:15.90 Xorg
 1957 demonlee  20   0  206556  28348  18504 S   0.3   0.5   0:00.26 ibus-x11
 2897 demonlee  20   0  874684  61296  44532 S   0.3   1.0   0:06.22 gnome-terminal-
19984 demonlee  20   0   20632   4036   3376 R   0.3   0.1   0:00.02 top
    1 root      20   0  169076  12952   8288 S   0.0   0.2   0:04.61 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.01 kthreadd
    3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
    4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
    6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-kblockd
    9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
   10 root      20   0       0      0      0 S   0.0   0.0   0:00.08 ksoftirqd/0
   11 root      20   0       0      0      0 I   0.0   0.0   0:03.60 rcu_sched
   12 root      rt   0       0      0      0 S   0.0   0.0   0:00.51 migration/0
   13 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_inject/0
   14 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
   15 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/1
   16 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_inject/1
   17 root      rt   0       0      0      0 S   0.0   0.0   0:00.89 migration/1
   18 root      20   0       0      0      0 S   0.0   0.0   0:00.09 ksoftirqd/1
   20 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/1:0H
   21 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/2
   22 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_inject/2
   23 root      rt   0       0      0      0 S   0.0   0.0   0:00.83 migration/2
   24 root      20   0       0      0      0 S   0.0   0.0   0:00.07 ksoftirqd/2
   26 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/2:0H-kblockd
   27 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/3
   28 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_inject/3
   29 root      rt   0       0      0      0 S   0.0   0.0   0:00.77 migration/3
   30 root      20   0       0      0      0 S   0.0   0.0   0:00.18 ksoftirqd/3
   32 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/3:0H-kblockd
   33 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kdevtmpfs
   34 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
   35 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_kthre
   36 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_rude_
   37 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_trace
   38 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kauditd
   39 root      20   0       0      0      0 S   0.0   0.0   0:00.03 khungtaskd
demonlee@demonlee-ubuntu:~$
```

最后，再补充一个知识点：使用ps命令查看进程时，会发现状态上面有其他符号，比如 S+、Z+ 等，如下所示，

```
demonlee@demonlee-ubuntu:~$
demonlee    1704  0.0  0.6 557904 37256 ?        Sl   05:01   0:00 /usr/libexec/goa-daemon
demonlee    1707  0.0  0.1 172652  6936 tty2     Ssl+ 05:01   0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/g
demonlee    1714  0.0  0.1 323388  9068 ?        Sl   05:01   0:00 /usr/libexec/goa-identity-service
demonlee    1720  0.1  1.3 1151880 80576 tty2    Sl+  05:01   1:05 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background non
demonlee    1723  0.0  0.1 325356  9016 ?        Ssl  05:01   0:02 /usr/libexec/gvfs-afc-volume-monitor
demonlee    1728  0.0  0.1 244336  6532 ?        Ssl  05:01   0:00 /usr/libexec/gvfs-mtp-volume-monitor
```

```
demonlee    1759  0.0  0.2 197052 14276 tty2     Sl+  05:01   0:00 /usr/libexec/gnome-session-binary --systemd --systemd --session=ubuntu
...
```

这个 + 是啥意思呢，其实manps中就有描述，只是我们从来都没认真看说明文档：

```
PROCESS STATE CODES
       Here are the different values that the s, stat and state output specifiers (header "STAT" or "S") will display to describe the state of a
       process:

               D    uninterruptible sleep (usually IO)
               I    Idle kernel thread
               R    running or runnable (on run queue)
               S    interruptible sleep (waiting for an event to complete)
               T    stopped by job control signal
               t    stopped by debugger during the tracing
               W    paging (not valid since the 2.6.xx kernel)
               X    dead (should never be seen)
               Z    defunct ("zombie") process, terminated but not reaped by its parent

       For BSD formats and when the stat keyword is used, additional characters may be displayed:

               <    high-priority (not nice to other users)
               N    low-priority (nice to other users)
               L    has pages locked into memory (for real-time and custom IO)
               s    is a session leader
               l    is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
               +    is in the foreground process group
```

编辑于 2022-02-13 17:13

Linux 进程 进程管理

What Is an Uninterruptible Process in Linux?

Last updated: November 30, 2022

Written by: Javier Lobato Perez (https://www.baeldung.com/linux/author/javierlobatoperez)

1. Overview

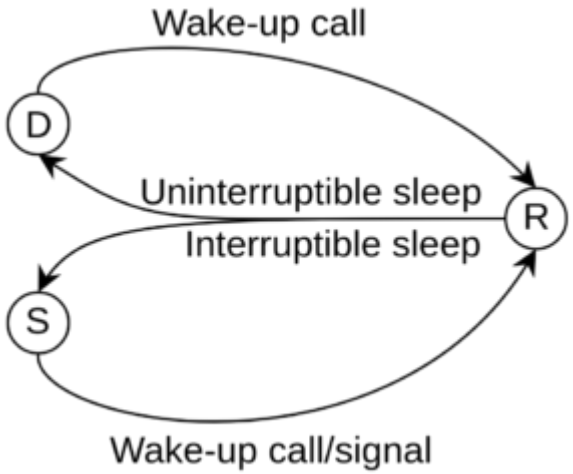
In this article, we'll present uninterruptible sleep processes. We'll begin by describing them together with their counterparts, interruptible sleep processes. Then, we'll discuss its identification and how we can manage them.

2. Interruptible and Uninterruptible Sleep

Before talking about uninterruptible processes, we need to discuss what interruptible processes are. Linux has different states for any given process (/linux/process-states). A process is running (or runnable) when it is in the *R* state.

Linux also has two ways of putting running processes *R* to sleep. **A process can be put into interruptible sleep *S* or uninterruptible sleep *D*.** In both cases, the process is waiting for an event to complete. On the one hand, **a process in the *S* state can return to the *R* state either by using an explicit wake-up call or by receiving a signal.** The error returned when interruptible system calls receive a signal is *EINTR* (https://man7.org/linux/man-pages/man3/errno.3.html).

On the other hand, the *D* state means that the process is ignoring signals. **Thus, the only way to go from uninterruptible sleep *D* to running *R* is with an explicit wake-up call:**



2.1. Stopping Processes in Interruptible Sleep With Signals

Processes can be in one of two different modes (/cs/user-kernel-modes): user mode and kernel mode. In user mode, we can send signals, but they are taken into account once the process returns from the kernel mode. **We cannot kill processes in kernel mode because this might corrupt data.**

Let's consider what happens with the system call *read( )* (https://man7.org/linux/man-pages/man2/read.2.html). When we use this system call, we don't know how much time the *read( )* operation will take. Thus, the scheduler puts the process into interruptible sleep so that the system can use the free resources for something else. The process created by *read( )* will be sleeping during the time it takes to complete the operation: it is blocked by the hardware, so it cannot continue. **If, during this sleeping time, we issue an asynchronous signal** (such as pressing *Ctrl+C*)**, we'll end the process.**

However, we cannot stop all system calls like this. This is the case for the processes that are in an uninterruptible sleep state – which require an explicit wake-up call!

2.2. Why Can't We Stop Processes in Uninterruptible Sleep?

As opposed to interruptible processes, when uninterruptible processes are in kernel mode, they won't accept interruption signals. The system call expects to be woken up exclusively by the process it depends on.

One common example of uninterruptible sleep is creating a folder with *mkdir* (/linux/file-manipulation#mkdir), which only enters uninterruptible sleep. Normally, *mkdir* has to perform some disk searches exclusively. If the I/O operation gets successfully completed, the process will continue. If the operation fails, the kernel will get a *SIGBUS* (or an alternative signal) to manage the parent process.

Nevertheless, if there is a network problem, *mkdir* might get stuck into an uninterruptible sleep state. The only way to leave this mode is with an explicit wake-up call from the parent call and not from the user.

2.3. Rationale Behind Uninterruptible Sleep

When reading the previous lines, it might seem that uninterruptible sleep is not that different from interruptible sleep. Moreover, it might not seem necessary at all to have the uninterruptible sleep state, as interruptible sleep looks like enhanced uninterruptible sleep where the user can terminate the system call.

However, the uninterruptible sleep state has some advantages over interruptible sleep. Programming system calls for interruptible sleep is significantly harder than for uninterruptible sleep. Writing code that handles interruptible sleep requires extensive code both in kernel mode (constantly checking for any wake-up call, and if so, handle it, clean the

memory and return) and in user mode (as the process should respond accordingly to the interrupted system call).

Making a system call go into uninterruptible sleep instead makes handling it smoother. Software developers rely on this type of sleep for short processes, to generate atomic calls, or to avoid restarting the main call.

2.4. Is Uninterruptible Sleep the Best Solution?

We have described interruptible sleep, uninterruptible sleep, and other states for Linux processes. However, there is a recent type of state known as *killable* (https://lwn.net/Articles/288056/) that might be useful when writing custom code, as it is a compromise between the two types of sleep states.

The killable state is based on the uninterruptible state but accepts *FATAL* signals that will interrupt the sleep state. Thus, if the call gets stuck, it can be killed. This state originated by observing that an application bug usually is not relevant if we kill the parent process. However, not all system calls implement this state yet. Those that don't still rely on uninterruptible sleep.

3. Manage Uninterruptible Processes

Now that we know how uninterruptible processes work, we'll discuss how to manage them. We need to identify, prevent, and kill them when they appear.

3.1. Identification of Uninterruptible Processes

We can check the state of the different processes of our machine with *ps* (/linux/ps-command):

```
$ ps a
PID    TTY    STAT  TIME   COMMAND
796    tty1   Sl    75:53  /usr/lib/Xorg -nolisten tcp :0 vt1 -keeptty -auth /tmp/serverauth.MKLkhKNSLR
813    tty1   Ss+   0:21   i3 -a --restart /run/user/1000/i3/restart-state.813
907    tty1   S     9:30   firefox
111189 pts/2  R+    0:00   ps avi /path/to/file/to/edit
```

We see that there are identifiers for each process state: *Ss+*, *Sl*, *R+*,... The state has four letters or symbols that describe its status within the system. **The first letter represents the status of the process, and it corresponds to the different letters previously mentioned (*S*, *R*, and *D*).**

We can also use *top* (/linux/top-command), which displays the process status in the column under *S*. We only see the first letter shown in the column *STAT* from *ps*:

```
$ top
...
PID    USER   PR  NI     VIRT     RES     SHR  S   %CPU   %MEM     TIME+   COMMAND
907     foo   20   0    14.0g    2.0g  879928  S    8.3   26.5   9:30.65   firefox
796     foo   20   0  1174872  746824  721272  S    3.7    9.5  75:53.34   /usr/lib/Xorg -nolisten tcp :0 vt1
-keeptty -auth /tmp/serverauth.MKLkhKNSLR
813     foo   20   0  2828196  436624   99976  S    2.7    5.6   0:21.69   i3 -a --restart
/run/user/1000/i3/restart-state.813
111189  foo   20   0  2781348  351432  117044  R    1.7    4.5   0:46.73   ps avi /path/to/file/to/edit
```

3.2. Preventing Processes to Enter Uninterruptible Sleep

We cannot completely avoid system calls entering uninterruptible sleep. As we discussed, they are something present in Linux, and programming with them is helpful. **From a user point of view, there is not much that we can do.** Most of the system calls that are uninterruptible happen instantaneously, so we usually don't observe processes in the *D* state.

However, the process can get stuck in the uninterruptible sleep phase. Apart from wrong I/O connections, we might also have buggy kernel drivers that freeze processes into uninterruptible sleep. **Thus, keeping our system drivers up to date and checking the release notes will help to minimize these problems.**

From a development point of view, we can try to reduce the number of uninterruptible system calls used in our code. Moreover, and depending on the depth of our development, we might want to use the *killable* state as an alternative to the uninterruptible sleep state.

3.3. Methods to Stop a Process in Uninterruptible Sleep

If we ever encounter a process into uninterruptible sleep, we need to check our hardware. If we encounter the issue when using network storage, it might be down, and the process is waiting for the server to recover. Once we know the driver that is causing the trouble, we can stop it (/linux/kill-background-process). We might need *rmmod* (https://linux.die.net/man/8/rmmod) to remove the module supporting the hardware device.

Another alternative is to use the parent process identifier (/linux/pid-tid-ppid#3-ppid-parent-process-identifier) of the process in uninterruptible sleep. **We can get the identifier of the parent process (known as PPID) and stop this process.** This is sufficient for cases where the parent process is an errant shell. Killing the parent process kills the child processes, which may trigger the explicit call required by the process in uninterruptible sleep.

Finally, the last solution when nothing else works is to suspend-to-disk or restart the system. We can try first to suspend-to-disk (also known as hibernate) and resume to see if this unfreezes the process in uninterruptible sleep. If this does not work, we have to restart the system. We might not be able to restart some systems, for example, a connected network device. In this case, we should attempt to unfreeze the process with the previous methods.

4. Conclusion

In this article, we've talked about processes that enter the uninterruptible sleep state, what this means, and how we can handle them.

Comments are closed on this article!

可以杀死的深度睡眠TASK_KILLABLE状态(最透彻一篇)

深度睡眠与浅度睡眠

众所周知， Linux 的进程睡眠有两种常规状态：

- TASK_INTERRUPTIBLE（浅度睡眠）：可以被等待的资源唤醒，也能被signal唤醒；
- TASK_UNINTERRUPTIBLE（深度睡眠）：可以被等待的资源唤醒，但是不能被signal唤醒。

简单来说，深度睡眠的进程必须等待资源来了才能醒，在此之前，甚至你给它发任何的信号，它都不可能醒来。

浅度睡眠的进程，则可以被信号唤醒，对于常规的键盘、串口、触摸屏等等这些I/O设备，显然符合此类模型。所以Linux内核的代码里面经常看到这样的代码模板，笔者在《Linux设备驱动开发详解》一书中也花了大篇幅解释如下模板：

```c
static ssize_t globalfifo_read(struct file *filp, char __user *buf,
                               size_t count, loff_t *ppos)
{
        int ret;
        struct globalfifo_dev *dev = container_of(filp->private_data,
                struct globalfifo_dev, miscdev);

        DECLARE_WAITQUEUE(wait, current);

        mutex_lock(&dev->mutex);
        add_wait_queue(&dev->r_wait, &wait);

        while (dev->current_len == 0) {
                if (filp->f_flags & O_NONBLOCK) {
                        ret = -EAGAIN;
                        goto out;
                }
                __set_current_state(TASK_INTERRUPTIBLE);
                mutex_unlock(&dev->mutex);

                schedule();
                if (signal_pending(current)) {
                        ret = -ERESTARTSYS;
                        goto out2;
                }

                mutex_lock(&dev->mutex);
        }

        if (count > dev->current_len)
                count = dev->current_len;

        if (copy_to_user(buf, dev->mem, count)) {
```

浅度睡眠

醒来第一件事往往是检测signal

调用__set_current_state(TASK_INTERRUPTIBLE)并schedule()出去的进程，醒来第一件事往往就是通过signal_pending(current)查看信号是否存在，如果存在，就跳出去处理信号，无需等待I/O的完成（大不了信号处理完了再重新read）。

TASK_INTERRUPTIBLE看起来很理想，不至于在I/O没完成的时候，连CTRL+C都不响应（当然也不会响应其他SIGIO、SIGUSR1等信号）。

那么，有的童鞋就会问，既然浅度睡眠这么好，那么还要TASK_UNINTERRUPTIBLE这种完全不响应信号的深度睡眠干什么?

深度睡眠不可避免

正在读本文的你，可能都有过这样的悲催经历，在NFS文件系统上面运行程序，但是NFS服务器挂了，你怎么都ctrl + c不掉那个进程，因为它就是个深度睡眠的场景。你徘徊，你迷茫，你问能不能直接都改为TASK_INTERRUPTIBLE，彻底删除TASK_UNINTERRUPTIBLE呢?

对此，祖师爷Linus的答复是：不可能。请看他2002年的邮件:

Date: Thu, 1 Aug 2002 20:23:18 GMT
Message-ID: <fa.n81d55v.fnah8e@ifi.uio.no>

On Thu, 1 Aug 2002, David Woodhouse wrote:
>
> torvalds@transmeta.com said:
> > They should be waiting in TASK_UNINTERRUPTIBLE, and we should add a
> > flag to distinguish between "increases load average" and "doesn't".
>
> The disadvantage of this approach is that it encourages people to be lazy
> and sleep with signals disabled, instead of implementing proper cleanup
> code.
>
> I'm more in favour of removing TASK_UNINTERRUPTIBLE entirely, or at least
> making people apply for a special licence to be permitted to use it :)

Can't do that.

Easy reason: there are tons of code sequences that _cannot_ take signals.
The only way to make a signal go away is to actually deliver it, and there
are documented interfaces that are guaranteed to complete without
delivering a signal. The trivial case is a disk read: real applications
break if you return partial results in order to handle signals in the
middle.

In short, this is not something that can be discussed. It's a cold fact, a
law of UNIX if you will.

There are enough reasons to discourage people from using uninterruptible
sleep ("this f*cking application won't die when the network goes down")
that I don't think this is an issue. We need to handle both cases, and
while we can expand on the two cases we have now, we can't remove them.

Linus

对于磁盘读等场景，如果读还没完成，就跳出去响应信号，application可能break，所以**深度睡眠必须存在是一个客观的残酷的现实（cold fact）**。

祖师爷还有更猛的一锤定音：

> > In short, this is not something that can be discussed. It's a cold fact, a
> > law of UNIX if you will.
>
> Any program setting up signal handlers should expext interrupted i/o,
> otherwise it's buggy.

Roman, THAT IS JUST NOT TRUE!

Go read the standards. Some IO is not interruptible. This is not something
I'm making up, and this is not something that can be discussed about. The
speed of light in vacuum is 'c', regardless of your own relative speed.
And file reads are not interruptible.

Linus

祖师爷没有点明为什么磁盘读的时候不应该跑用户态去执行信号处理函数，为什么引发application break。我理解其中的一个场景如下：Linux对于代码段、数据段、堆和栈都通常依赖demanding page在发生page fault的时候从磁盘swap进来的，从而导致磁盘读的行为。在这个过程中，如果我们执行浅度睡眠并响应信号而跳过去执行应用程序代码段设置的信号处理函数，则此信号处理函数的执行可能再次因为swap in的需求引发进一步的磁盘读，造成double page fault的场景。磁盘的读很大程度上不一定是read系统调用引发的，考虑到代码段、数据段、堆和栈的往往是发生了page fault后才去从磁盘swap进来。**磁盘有其特殊性，在Linux这样的操作系统里面，磁盘某种意义上是"内存"。**

但是，如果响应信号后，哪怕application break都已经无所谓了呢？如果我们的目的干脆就是发一个致命的信号，譬如杀死应用的信号(SIGKILL)，那么application break这个就显得无关紧要了，**因为我们本身就不打算继续玩下去了！**这样就使得深度睡眠的进程，还可以被杀死，妈妈再也不用担心NFS服务器挂了后，我痛苦，我孤独，我精分了!

可杀的深度睡眠

Linux因此推出了一个特殊的深度睡眠状态，叫做

- TASK_KILLABLE（可杀的深度睡眠）：可以被等到的资源唤醒，不能被常规信号唤醒，但是可以被致命信号唤醒，**醒后即死。**

TASK_KILLABLE状态的定义是：

#define TASK_KILLABLE (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)

所以它显然是属于TASK_UNINTERRUPTIBLE的，只是可以被TASK_WAKEKILL。

什么叫致命信号呢？talk is cheap，show me the code。

```
static inline int __fatal_signal_pending(struct task_struct *p)
{
        return unlikely(sigismember(&p->pending.signal, SIGKILL));
}
```

所以，足够致命的信号就是SIGKILL。SIGKILL何许人也，就是传说中的信号9，无法阻挡无法被应用覆盖的终极杀器：

```
barry@barryUbuntu:~/develop/linux$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

仅仅从这个代码可以看出来，只有信号9才属于fatal signals。那么是不是只有信号9，才可以杀死TASK_KILLABLE的进程，信号2（CTRL+C）是否无能为力呢？

猜想再多，不如玩一个真实的代码，我们下面来改造下，把globalfifo.c的read改造为TASK_KILLABLE。

```
diff --git a/kernel/drivers/globalfifo/ch12/globalfifo.c b/kernel/drivers/globalfifo/ch12/globalfifo.c
index 6d21488..89fae6c 100644
--- a/kernel/drivers/globalfifo/ch12/globalfifo.c
+++ b/kernel/drivers/globalfifo/ch12/globalfifo.c
@@ -118,11 +118,12 @@ static ssize_t globalfifo_read(struct file *filp, char __user *buf,
                        ret = -EAGAIN;
                        goto out;
                }
-               __set_current_state(TASK_INTERRUPTIBLE);
+               __set_current_state(TASK_KILLABLE);
                mutex_unlock(&dev->mutex);

                schedule();
-               if (signal_pending(current)) {
+               if (fatal_signal_pending(current)) {
+                       pr_info("wake-up by fatal signal %llx\n", current->pending.signal);
                        ret = -ERESTARTSYS;
                        goto out2;
                }
@@ -195,7 +196,7 @@ static ssize_t globalfifo_write(struct file *filp, const char __user *buf,
                printk(KERN_INFO "written %d bytes(s),current_len:%d\n", count,
                       dev->current_len);

-               wake_up_interruptible(&dev->r_wait);
+               wake_up(&dev->r_wait);

                if (dev->async_queue) {
                        kill fasync(&dev->async queue, SIGIO, POLL IN);
```

加载这个driver后，我们来读取它：

insmod globalfifo.ko

insmod globalfifo-dev.ko

cat /dev/globalfifo

这个时候，我们ps命令看一下，可以清楚到看到cat进程处于D状态：

root 7658 0.0 0.0 16800 752 pts/1 **D+** 19:21 0:00 cat /dev/globalfifo

从前面的代码可以看出，CTRL+C是不应该可以杀死这个cat进程的，因为它不是SIGKILL。但是我们来实际测试一下：

cat /dev/globalfifo

^C

#

实际却是可以杀死！！！

我们查看一下我们加的那个内核打印代码，看一下signal pending的情况：

dmesg

[4670.082548] wake-up by fatal signal 100

明明我们发的是信号2，但是被置上的就是信号9(0x100的1对应SIGKILL的位)。这里发生了神奇的化学反应！！！

这踏马到底是怎么回事？不是一定致命的信号2，为什么转化为了最最致命的信号9呢？

信号2是如何转化为信号9的？

这个时候我们重点关注kernel/signal.c内核代码中的complete_signal()函数：

```
if (sig_fatal(p, sig) &&
    !(signal->flags & (SIGNAL_UNKILLABLE | SIGNAL_GROUP_EXIT)) &&
    !sigismember(&t->real_blocked, sig) &&
    (sig == SIGKILL || !t->ptrace)) {
        /*
         * This signal will be fatal to the whole group.
         */
        if (!sig_kernel_coredump(sig)) {
                /*
                 * Start a group exit and wake everybody up.
                 * This way we don't have other threads
                 * running and doing things after a slower
                 * thread has the fatal signal pending.
                 */
                signal->flags = SIGNAL_GROUP_EXIT;
                signal->group_exit_code = sig;
                signal->group_stop_count = 0;
                t = p;
                do {
                        task_clear_jobctl_pending(t, JOBCTL_PENDING_MASK);
                        sigaddset(&t->pending.signal, SIGKILL);
                        signal_wake_up(t, 1);
                } while_each_thread(p, t);
                return;
        }
}
```

实际上，当Linux内核发现进程（线程组）收到了一个sig_fatal()的信号的时候，会给这个进程中的每个线程人为地插入一个SIGKILL信号，这个从while_each_thread循环可以看出。

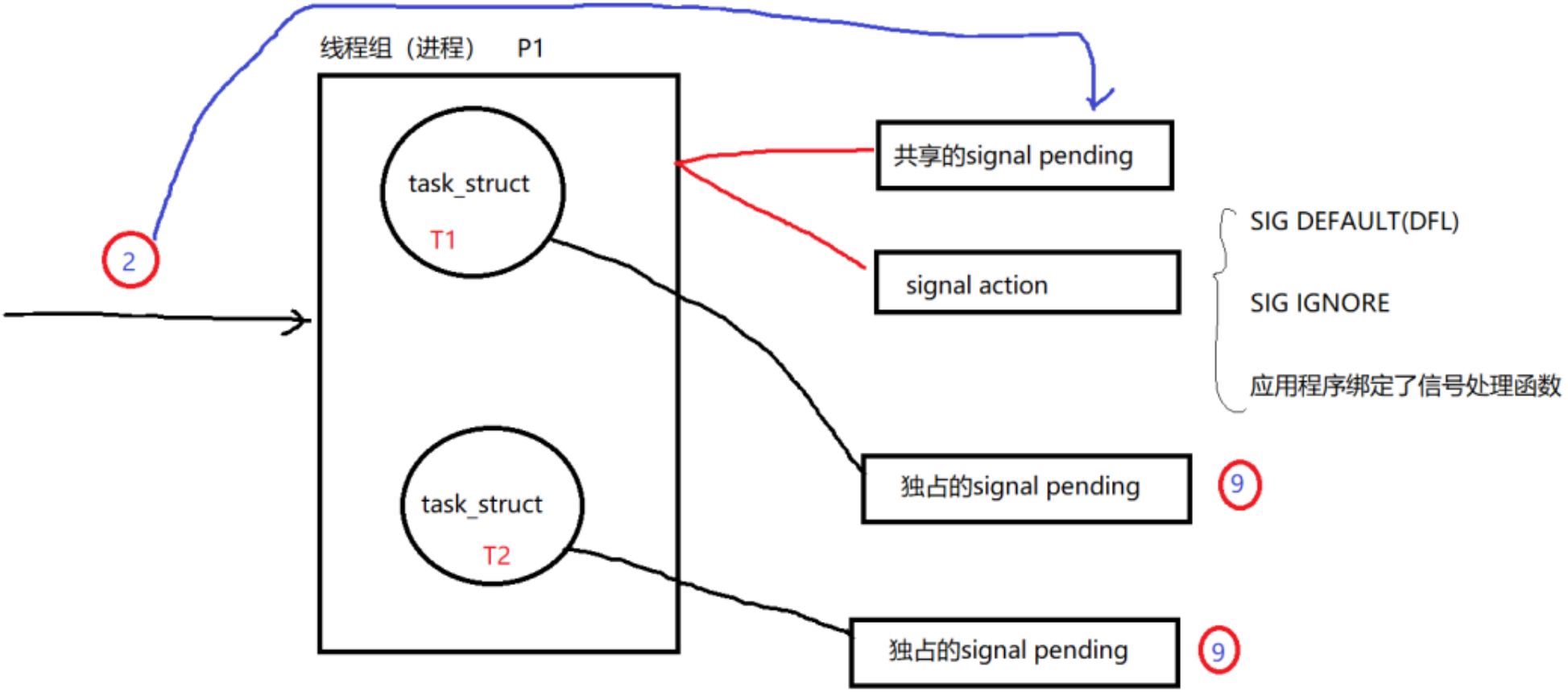
sig_fatal()和fatal_signal_pending()不是一个概念。我们看看sig_fatal()的代码：

```
#define sig_fatal(t, signr) \
        (!siginmask(signr, SIG_KERNEL_IGNORE_MASK|SIG_KERNEL_STOP_MASK) && \
         (t)->sighand->action[(signr)-1].sa.sa_handler == SIG_DFL)
```

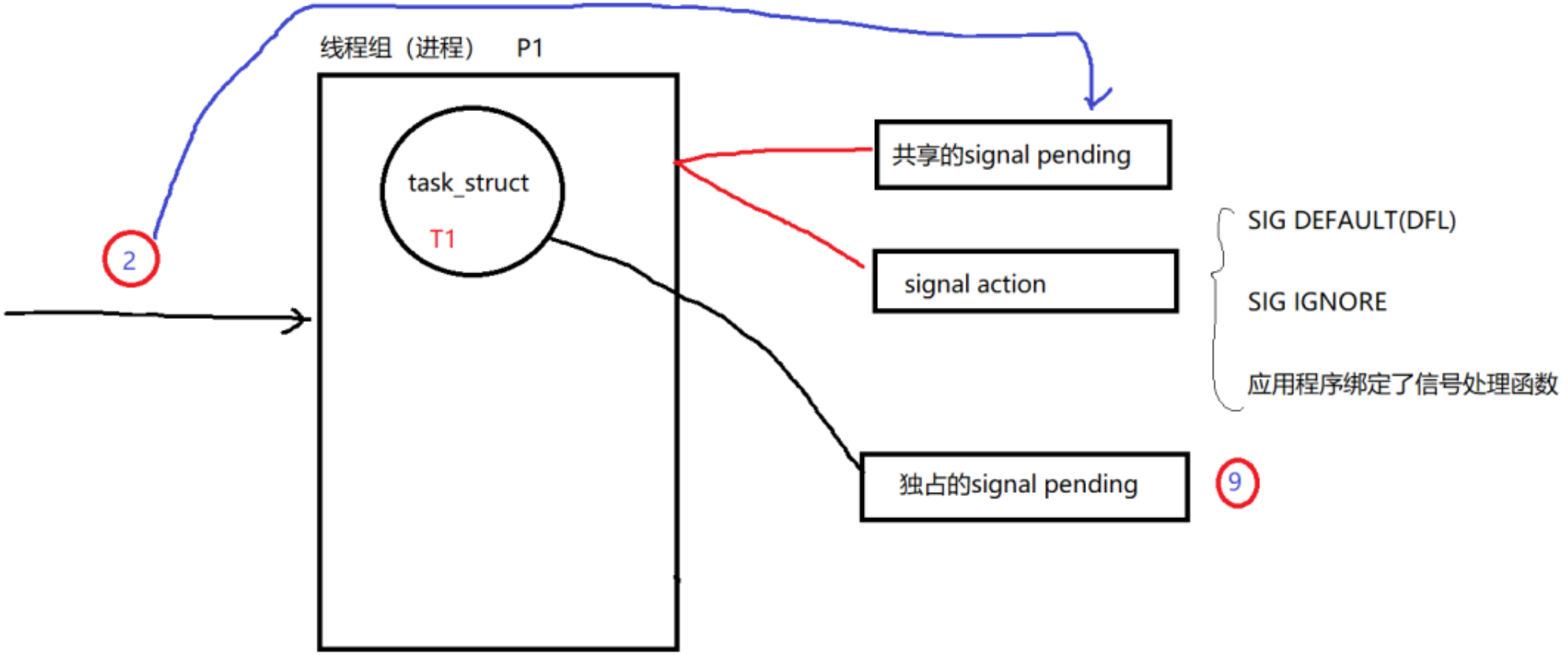
基本上，一个信号的行为如果是缺省的（SIG_DFL），它又没有被忽略，那么它就是满足sig_fatal()条件的。

如下图，流程大概是：

当我们给进程P1（假设内部有线程T1和T2，那么每个线程会有个tast_struct）发送信号2,这个2会填入T1和T2共享的进程级signal pending，由于我们对信号2没有绑定和忽略而是采用了默认行为，于是导致sig_fatal()条件满足。内核就会在T1和T2的各自独占的一份signal pending里面填入9，从而刺激fatal_signal_pending()条件的满足。



有的童鞋说，如果我的进程只有一个线程呢？那去掉上图中的T2以及T2独占的signal pending框即可：



为了进行验证，我们不再使用cat。而是自己写个app去访问globalfifo，而在此app里面修改信号2的行为：

```c
void sigint(int sig)
{
        printf("got the SIGINT signal...\n");
}

int main(void)
{
        char buf[100];
        int fd=open("/dev/globalfifo",O_RDONLY);

        signal(2, sigint);

        if(fd==-1){
                perror("cannot open the FIFO");
                return -11;
        }

        read(fd, buf, 10);

        return 0;
}
```

我们通过signal(2, sigint)给信号2绑定了信号处理函数sigint()，这个时候read(fd, buf, 10)引发TASK_KILLABLE睡眠，我们无论怎么kill -2，都杀不死上面这个app。2到9的转化过程不再发生。

下面的修改也可达到类似效果：

```c
int main(void)
{
        char buf[100];
        int fd=open("/dev/globalfifo",O_RDONLY);

        signal(2, SIG_IGN);

        if(fd==-1){
                perror("cannot open the FIFO");
                return -11;
        }

        read(fd, buf, 10);

        return 0;
}
```

上面我们是把信号2进行了SIG_IGN的忽略处理。

不仅信号2是这样的，其他的很多信号也类似，比如SIGHUP、SIGIO、SIGTERM、SIGPIPE等都可以在没有绑定和忽略的情况下，转化为信号9。但是SIGCHLD显然不一样，因为SIGCHLD默认就是忽略的。

(END)

TASK_KILLABLE

By **Jonathan Corbet**
July 1, 2008

Like most versions of Unix, Linux has two fundamental ways in which a process can be put to sleep. A process which is placed in the `TASK_INTERRUPTIBLE` state will sleep until either (1) something explicitly wakes it up, or (2) a non-masked signal is received. The `TASK_UNINTERRUPTIBLE` state, instead, ignores signals; processes in that state will require an explicit wakeup before they can run again.

There are advantages and disadvantages to each type of sleep. Interruptible sleeps enable faster response to signals, but they make the programming harder. Kernel code which uses interruptible sleeps must always check to see whether it woke up as a result of a signal, and, if so, clean up whatever it was doing and return `-EINTR` back to user space. The user-space side, too, must realize that a system call was interrupted and respond accordingly; not all user-space programmers are known for their diligence in this regard. Making a sleep uninterruptible eliminates these problems, but at the cost of being, well, uninterruptible. If the expected wakeup event does not materialize, the process will wait forever and there is usually nothing that anybody can do about it short of rebooting the system. This is the source of the dreaded, unkillable process which is shown to be in the "D" state by `ps`.

Given the highly obnoxious nature of unkillable processes, one would think that interruptible sleeps should be used whenever possible. The problem with that idea is that, in many cases, the introduction of interruptible sleeps is likely to lead to application bugs. As recently noted by Alan Cox:

> Unix tradition (and thus almost all applications) believe file store writes to be non signal interruptible. It would not be safe or practical to change that guarantee.

So it would seem that we are stuck with the occasional blocked-and-immortal process forever.

Or maybe not. A while back, Matthew Wilcox realized that many of these concerns about application bugs do not really apply if the application is about to be killed anyway. It does not matter if the developer thought about the possibility of an interrupted system call if said system call is doomed to never return to user space. So Matthew created a new sleeping state, called `TASK_KILLABLE`; it behaves like `TASK_UNINTERRUPTIBLE` with the exception that fatal signals will interrupt the sleep.

With `TASK_KILLABLE` comes a new set of primitives for waiting for events and acquiring locks:

```
int wait_event_killable(wait_queue_t queue, condition);
long schedule_timeout_killable(signed long timeout);
int mutex_lock_killable(struct mutex *lock);
int wait_for_completion_killable(struct completion *comp);
int down_killable(struct semaphore *sem);
```

For each of these functions, the return value will be zero for a normal, successful return, or a negative error code in case of a fatal signal. In the latter case, kernel code should clean up and return, enabling the process to be killed.

The TASK_KILLABLE patch was merged for the 2.6.25 kernel, but that does not mean that the unkillable process problem has gone away. The number of places in the kernel (as of 2.6.26-rc8) which are actually using this new state is quite small - as in, one need not worry about running out of fingers while counting them. The NFS client code has been converted, which can only be a welcome development. But there are very few other uses of TASK_KILLABLE, and none at all in device drivers, which is often where processes get wedged.

It can take time for a new API to enter widespread use in the kernel, especially when it supplements an existing functionality which works well enough most of the time. Additionally, the benefits of a mass conversion of existing code to killable sleeps are not entirely clear. But there are almost certainly places in the kernel which could be improved by this change, if users and developers could identify the spots where processes get hung. It also makes sense to use killable sleeps in new code unless there is some pressing reason to disallow interruptions altogether.

Index entries for this article
Kernel Scheduler