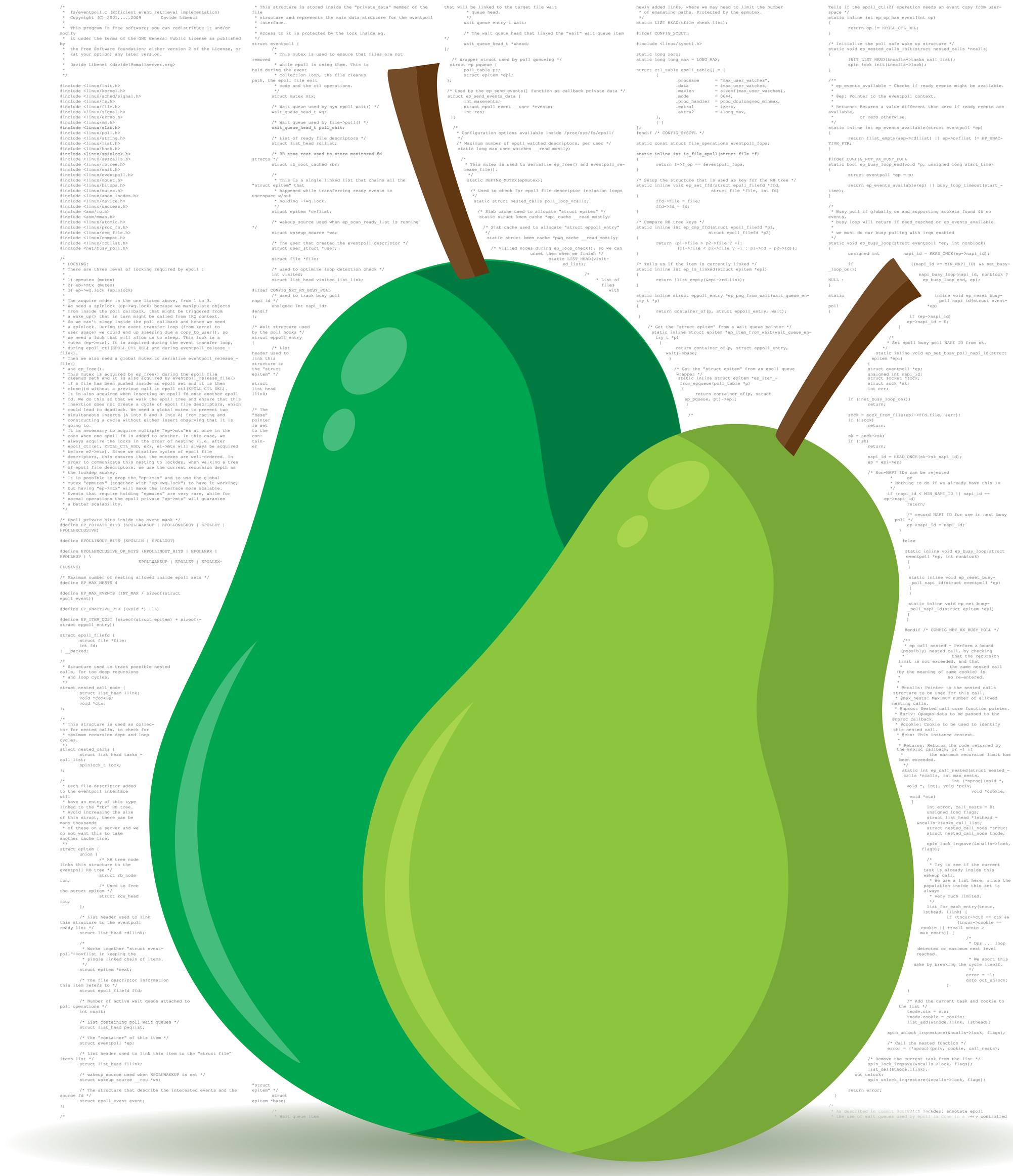
Pairing Station



```
/*
 *  fs/eventpoll.c (Efficient event retrieval implementation)
 *  Copyright (C) 2001,...,2009        Davide Libenzi
 *
 *  This program is free software; you can redistribute it and/or
modify
 *  it under the terms of the GNU General Public License as published
by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  Davide Libenzi <davidel@xmailserver.org>
 *
 */

#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/sched/signal.h>
#include <linux/fs.h>
#include <linux/file.h>
#include <linux/signal.h>
#include <linux/errno.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <linux/poll.h>
#include <linux/string.h>
#include <linux/list.h>
#include <linux/hash.h>
#include <linux/spinlock.h>
#include <linux/syscalls.h>
#include <linux/rbtree.h>
#include <linux/wait.h>
#include <linux/eventpoll.h>
#include <linux/mount.h>
#include <linux/bitops.h>
#include <linux/mutex.h>
#include <linux/anon_inodes.h>
#include <linux/device.h>
#include <linux/uaccess.h>
#include <asm/io.h>
#include <asm/mman.h>
#include <linux/atomic.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/compat.h>
#include <linux/rculist.h>
#include <net/busy_poll.h>

/*
 * LOCKING:
 * There are three level of locking required by epoll :
 *
 * 1) epmutex (mutex)
 * 2) ep->mtx (mutex)
 * 3) ep->wq.lock (spinlock)
 *
 * The acquire order is the one listed above, from 1 to 3.
 * We need a spinlock (ep->wq.lock) because we manipulate objects
 * from inside the poll callback, that might be triggered from
 * a wake_up() that in turn might be called from IRQ context.
 * So we can't sleep inside the poll callback and hence we need
 * a spinlock. During the event transfer loop (from kernel to
 * user space) we could end up sleeping due a copy_to_user(), so
 * we need a lock that will allow us to sleep. This lock is a
 * mutex (ep->mtx). It is acquired during the event transfer loop,
 * during epoll_ctl(EPOLL_CTL_DEL) and during eventpoll_release_-
file().
 * Then we also need a global mutex to serialize eventpoll_release_-
file()
 * and ep_free().
 * This mutex is acquired by ep_free() during the epoll file
 * cleanup path and it is also acquired by eventpoll_release_file()
 * if a file has been pushed inside an epoll set and it is then
 * close()d without a previous call to epoll_ctl(EPOLL_CTL_DEL).
 * It is also acquired when inserting an epoll fd onto another epoll
 * fd. We do this so that we walk the epoll tree and ensure that this
 * insertion does not create a cycle of epoll file descriptors, which
 * could lead to deadlock. We need a global mutex to prevent two
 * simultaneous inserts (A into B and B into A) from racing and
 * constructing a cycle without either insert observing that it is
 * going to.
 * It is necessary to acquire multiple "ep->mtx"es at once in the
 * case when one epoll fd is added to another. In this case, we
 * always acquire the locks in the order of nesting (i.e. after
 * epoll_ctl(e1, EPOLL_CTL_ADD, e2), e1->mtx will always be acquired
 * before e2->mtx). Since we disallow cycles of epoll file
 * descriptors, this ensures that the mutexes are well-ordered. In
 * order to communicate this nesting to lockdep, when walking a tree
 * of epoll file descriptors, we use the current recursion depth as
 * the lockdep subkey.
 * It is possible to drop the "ep->mtx" and to use the global
 * mutex "epmutex" (together with "ep->wq.lock") to have it working,
 * but having "ep->mtx" will make the interface more scalable.
 * Events that require holding "epmutex" are very rare, while for
 * normal operations the epoll private "ep->mtx" will guarantee
 * a better scalability.
 */

/* Epoll private bits inside the event mask */
#define EP_PRIVATE_BITS (EPOLLWAKEUP | EPOLLONESHOT | EPOLLET |
EPOLLEXCLUSIVE)

#define EPOLLINOUT_BITS (EPOLLIN | EPOLLOUT)

#define EPOLLEXCLUSIVE_OK_BITS (EPOLLINOUT_BITS | EPOLLERR |
EPOLLHUP | \
                                EPOLLWAKEUP | EPOLLET | EPOLLEX-
CLUSIVE)

/* Maximum number of nesting allowed inside epoll sets */
#define EP_MAX_NESTS 4

#define EP_MAX_EVENTS (INT_MAX / sizeof(struct
epoll_event))

#define EP_UNACTIVE_PTR ((void *) -1L)

#define EP_ITEM_COST (sizeof(struct epitem) + sizeof(-
struct eppoll_entry))

struct epoll_filefd {
        struct file *file;
        int fd;
} __packed;

/*
 * Structure used to track possible nested
calls, for too deep recursions
 * and loop cycles.
 */
struct nested_call_node {
        struct list_head llink;
        void *cookie;
        void *ctx;
};

/*
 * This structure is used as collec-
tor for nested calls, to check for
 * maximum recursion dept and loop
cycles.
 */
struct nested_calls {
        struct list_head tasks_-
call_list;
        spinlock_t lock;
};

/*
 * Each file descriptor added
to the eventpoll interface
will
 * have an entry of this type
linked to the "rbr" RB tree.
 * Avoid increasing the size
of this struct, there can be
many thousands
 * of these on a server and we
do not want this to take
another cache line.
 */
struct epitem {
        union {
                /* RB tree node
links this structure to the
eventpoll RB tree */
                struct rb_node
rbn;
                /* Used to free
the struct epitem */
                struct rcu_head
rcu;
        };

        /* List header used to link
this structure to the eventpoll
ready list */
        struct list_head rdllink;

        /*
         * Works together "struct event-
poll"->ovflist in keeping the
         * single linked chain of items.
         */
        struct epitem *next;

        /* The file descriptor information
this item refers to */
        struct epoll_filefd ffd;

        /* Number of active wait queue attached to
poll operations */
        int nwait;

        /* List containing poll wait queues */
        struct list_head pwqlist;

        /* The "container" of this item */
        struct eventpoll *ep;

        /* List header used to link this item to the "struct file"
items list */
        struct list_head fllink;

        /* wakeup_source used when EPOLLWAKEUP is set */
        struct wakeup_source __rcu *ws;

        /* The structure that describe the interested events and the
source fd */
        struct epoll_event event;
};

/*
 * This structure is stored inside the "private_data" member of the
file
 * structure and represents the main data structure for the eventpoll
 * interface.
 *
 * Access to it is protected by the lock inside wq.
 */
struct eventpoll {
        /*
         * This mutex is used to ensure that files are not
removed
         * while epoll is using them. This is
held during the event
         * collection loop, the file cleanup
path, the epoll file exit
         * code and the ctl operations.
         */
        struct mutex mtx;

        /* Wait queue used by sys_epoll_wait() */
        wait_queue_head_t wq;

        /* Wait queue used by file->poll() */
        wait_queue_head_t poll_wait;

        /* List of ready file descriptors */
        struct list_head rdllist;

        /* RB tree root used to store monitored fd
structs */
        struct rb_root_cached rbr;

        /*
         * This is a single linked list that chains all the
"struct epitem" that
         * happened while transferring ready events to
userspace w/out
         * holding ->wq.lock.
         */
        struct epitem *ovflist;

        /* wakeup_source used when ep_scan_ready_list is running
*/
        struct wakeup_source *ws;

        /* The user that created the eventpoll descriptor */
        struct user_struct *user;

        struct file *file;

        /* used to optimize loop detection check */
        int visited;
        struct list_head visited_list_link;

#ifdef CONFIG_NET_RX_BUSY_POLL
        /* used to track busy poll
napi_id */
        unsigned int napi_id;
#endif
};

/* Wait structure used
by the poll hooks */
struct eppoll_entry
{
        /* List
header used to
link this
structure to
the "struct
epitem" */

struct
list_head
llink;


/* The
"base"
pointer
is set
to the
con-
tain-
er
that will be linked to the target file wait
         * queue head.
         */
        wait_queue_entry_t wait;

        /* The wait queue head that linked the "wait" wait queue item
*/
        wait_queue_head_t *whead;
};

  /* Wrapper struct used by poll queueing */
  struct ep_pqueue {
        poll_table pt;
        struct epitem *epi;
};

/* Used by the ep_send_events() function as callback private data */
struct ep_send_events_data {
        int maxevents;
        struct epoll_event __user *events;
        int res;
};

  /*
   * Configuration options available inside /proc/sys/fs/epoll/
   */
  /* Maximum number of epoll watched descriptors, per user */
  static long max_user_watches __read_mostly;

  /*
   * This mutex is used to serialize ep_free() and eventpoll_re-
lease_file().
   */
  static DEFINE_MUTEX(epmutex);

    /* Used to check for epoll file descriptor inclusion loops
     */
    static struct nested_calls poll_loop_ncalls;

      /* Slab cache used to allocate "struct epitem" */
      static struct kmem_cache *epi_cache __read_mostly;

      /* Slab cache used to allocate "struct eppoll_entry"
       */
      static struct kmem_cache *pwq_cache __read_mostly;

      /* Visited nodes during ep_loop_check(), so we can
                  unset them when we finish */
                static LIST_HEAD(visit-
ed_list);

                                /*
                                 * List of
                                 files
                                  with
newly added links, where we may need to limit the number
 * of emanating paths. Protected by the epmutex.
 */
static LIST_HEAD(tfile_check_list);

#ifdef CONFIG_SYSCTL

#include <linux/sysctl.h>

static long zero;
static long long_max = LONG_MAX;

struct ctl_table epoll_table[] = {
        {
                .procname       = "max_user_watches",
                .data           = &max_user_watches,
                .maxlen         = sizeof(max_user_watches),
                .mode           = 0644,
                .proc_handler   = proc_doulongvec_minmax,
                .extra1         = &zero,
                .extra2         = &long_max,
        },
        { }
};
#endif /* CONFIG_SYSCTL */

static const struct file_operations eventpoll_fops;

static inline int is_file_epoll(struct file *f)
{
        return f->f_op == &eventpoll_fops;
}

/* Setup the structure that is used as key for the RB tree */
static inline void ep_set_ffd(struct epoll_filefd *ffd,
                              struct file *file, int fd)
{
        ffd->file = file;
        ffd->fd = fd;
}

/* Compare RB tree keys */
static inline int ep_cmp_ffd(struct epoll_filefd *p1,
                             struct epoll_filefd *p2)
{
        return (p1->file > p2->file ? +1:
                (p1->file < p2->file ? -1 : p1->fd - p2->fd));
}

/* Tells us if the item is currently linked */
static inline int ep_is_linked(struct epitem *epi)
{
        return !list_empty(&epi->rdllink);
}

static inline struct eppoll_entry *ep_pwq_from_wait(wait_queue_en-
try_t *p)
{
        return container_of(p, struct eppoll_entry, wait);
}

  /* Get the "struct epitem" from a wait queue pointer */
  static inline struct epitem *ep_item_from_wait(wait_queue_en-
try_t *p)
    {
        return container_of(p, struct eppoll_entry,
wait)->base;
    }

    /* Get the "struct epitem" from an epoll queue
wrapper */
    static inline struct epitem *ep_item_-
from_epqueue(poll_table *p)
    {
        return container_of(p, struct
ep_pqueue, pt)->epi;
    }

    /*
Tells if the epoll_ctl(2) operation needs an event copy from user-
space */
static inline int ep_op_has_event(int op)
{
        return op != EPOLL_CTL_DEL;
}

/* Initialize the poll safe wake up structure */
static void ep_nested_calls_init(struct nested_calls *ncalls)
{
        INIT_LIST_HEAD(&ncalls->tasks_call_list);
        spin_lock_init(&ncalls->lock);
}

/**
 * ep_events_available - Checks if ready events might be available.
 *
 * @ep: Pointer to the eventpoll context.
 *
 * Returns: Returns a value different than zero if ready events are
available,
 *          or zero otherwise.
 */
static inline int ep_events_available(struct eventpoll *ep)
{
        return !list_empty(&ep->rdllist) || ep->ovflist != EP_UNAC-
TIVE_PTR;
}

#ifdef CONFIG_NET_RX_BUSY_POLL
static bool ep_busy_loop_end(void *p, unsigned long start_time)
{
        struct eventpoll *ep = p;

        return ep_events_available(ep) || busy_loop_timeout(start_-
time);
}

/*
 * Busy poll if globally on and supporting sockets found && no
events,
 * busy loop will return if need_resched or ep_events_available.
 *
 * we must do our busy polling with irqs enabled
 */
static void ep_busy_loop(struct eventpoll *ep, int nonblock)
{
        unsigned int        napi_id = READ_ONCE(ep->napi_id);

        if                  ((napi_id >= MIN_NAPI_ID) && net_busy-
_loop_on())
                              napi_busy_loop(napi_id, nonblock ?
NULL :
                                             ep_busy_loop_end, ep);
}

static                        inline void ep_reset_busy-
                               _poll_napi_id(struct event-
poll
                             *ep)
{
                          if (ep->napi_id)
                        ep->napi_id = 0;
                    }

                    /*
                     * Set epoll busy poll NAPI ID from sk.
                     */
                    static inline void ep_set_busy_poll_napi_id(struct
                    epitem *epi)
                    {
                    struct eventpoll *ep;
                    unsigned int napi_id;
                    struct socket *sock;
                    struct sock *sk;
                    int err;

                    if (!net_busy_loop_on())
                            return;

                    sock = sock_from_file(epi->ffd.file, &err);
                    if (!sock)
                            return;

                    sk = sock->sk;
                    if (!sk)
                            return;

                    napi_id = READ_ONCE(sk->sk_napi_id);
                    ep = epi->ep;

                    /* Non-NAPI IDs can be rejected
                     *      or
                     * Nothing to do if we already have this ID
                     */
                    if (napi_id < MIN_NAPI_ID || napi_id ==
                    ep->napi_id)
                            return;

                    /* record NAPI ID for use in next busy
                    poll */
                    ep->napi_id = napi_id;
                }

                #else

                static inline void ep_busy_loop(struct
                eventpoll *ep, int nonblock)
                {
                }

                static inline void ep_reset_busy-
                _poll_napi_id(struct eventpoll *ep)
                {
                }

                static inline void ep_set_busy-
                _poll_napi_id(struct epitem *epi)
                {
                }

                #endif /* CONFIG_NET_RX_BUSY_POLL */

                /**
                 * ep_call_nested - Perform a bound
                (possibly) nested call, by checking
                 *                  that the recursion
                limit is not exceeded, and that
                 *                  the same nested call
                (by the meaning of same cookie) is
                 *                  no re-entered.
                 *
                 * @ncalls: Pointer to the nested_calls
                structure to be used for this call.
                 * @max_nests: Maximum number of allowed
                nesting calls.
                 * @nproc: Nested call core function pointer.
                 * @priv: Opaque data to be passed to the
                @nproc callback.
                 * @cookie: Cookie to be used to identify
                this nested call.
                 * @ctx: This instance context.
                 *
                 * Returns: Returns the code returned by
                the @nproc callback, or -1 if
                 *          the maximum recursion limit has
                been exceeded.
                 */
                static int ep_call_nested(struct nested_-
                calls *ncalls, int max_nests,
                                          int (*nproc)(void *,
                void *, int), void *priv,
                                                       void *cookie,
                void *ctx)
                {
                        int error, call_nests = 0;
                        unsigned long flags;
                        struct list_head *lsthead =
                      &ncalls->tasks_call_list;
                        struct nested_call_node *tncur;
                        struct nested_call_node tnode;

                        spin_lock_irqsave(&ncalls->lock,
                      flags);

                        /*
                         * Try to see if the current
                       task is already inside this
                       wakeup call.
                         * We use a list here, since the
                       population inside this set is
                       always
                         * very much limited.
                         */
                        list_for_each_entry(tncur,
                       lsthead, llink) {
                                if (tncur->ctx == ctx &&
                                    (tncur->cookie ==
                        cookie || ++call_nests >
                        max_nests)) {
                                        /*
                                         * Ops ... loop
                          detected or maximum nest level
                          reached.
                                         * We abort this
                        wake by breaking the cycle itself.
                                         */
                                        error = -1;
                                        goto out_unlock;
                                }
                        }

                        /* Add the current task and cookie to
                       the list */
                        tnode.ctx = ctx;
                        tnode.cookie = cookie;
                        list_add(&tnode.llink, lsthead);

                      spin_unlock_irqrestore(&ncalls->lock, flags);

                      /* Call the nested function */
                      error = (*nproc)(priv, cookie, call_nests);

                    /* Remove the current task from the list */
                    spin_lock_irqsave(&ncalls->lock, flags);
                    list_del(&tnode.llink);
                out_unlock:
                    spin_unlock_irqrestore(&ncalls->lock, flags);

                return error;
                }

                /*
                 * As described in commit 0ccf831cb lockdep: annotate epoll
                 * the use of wait queues used by epoll is done in a very controlled
```

```
var codeQuality = Math.Pow(2, awesome);
```