```c
/*
 * This file contains the source code for the curse system call.
 *
 * [The functions used by the system call are sourced below it.]
 *
 */
#include <linux/syscalls.h>
#ifdef CONFIG_CURSES

#include <linux/types.h>      /*Sentinels prevent multiple inclusion.*/
#include <linux/spinlock.h>
#include <linux/rcupdate.h>
#include <linux/namei.h>

#include <curse/curse.h>
#include <curse/curse_types.h>

#define CURSE_SYSTEM 0
#define CURSE_TARGETED 1
#define CURSE_REMOTE 2

//=====External declarations.
extern int max_curse_no;
extern struct curse_list_entry *curse_full_list;

//=====Various wrapper functions.
/*This function returns the bitmask for the specified normalized curse index.*/
inline uint64_t bitmask_from_no (int  a_c_id)
{
   return curse_list_pointer[a_c_id].curse_bit;
}

/*This macro expands to the requested field of the requested element of curse_list_pointer
array.*/
#define CURSE_FIELD(el, field) (curse_list_pointer[(el)].field)

/*This function checks if current is allowed to change the state of the target proc.*/
static int check_permissions (pid_t target, int type)
{
   struct task_struct *foreign_task;
   const struct cred *foreign_c = NULL, *local_c = NULL;
   uint8_t local_curse_perms;
   uint8_t foreign_curse_perms;
   int ret = -EINVAL;
   unsigned long spinflags;

   spin_lock_irqsave(&((current->curse_data).protection), spinflags);
   local_curse_perms = current->curse_data.permissions;
   spin_unlock_irqrestore(&((current->curse_data).protection), spinflags);
   local_c = get_current_cred();

   switch(type) {
      case CURSE_SYSTEM:
         ret = -EPERM;
         if ((local_c->euid == 0) && (local_curse_perms & _SU_ACTIVE_PERM))
            ret = 1;
         goto out_with_local;
      case CURSE_REMOTE:
      case CURSE_TARGETED:
         ret = -ESRCH;     //FIXME: Sanity check.
         rcu_read_lock();
         foreign_task = find_task_by_vpid(target);
         rcu_read_unlock();
         if (!foreign_task)
            goto out;

         ret = -EINVAL;    //FIXME: Sanity check.
         foreign_c = get_task_cred(foreign_task);

         if (!foreign_c)
            goto out_with_local;
         /* am i root or sudo?? */
         /* do we belong to the same effective user?*/

         spin_lock_irqsave(&((foreign_task->curse_data).protection), spinflags);
         foreign_curse_perms = foreign_task->curse_data.permissions;
         spin_unlock_irqrestore(&((foreign_task->curse_data).protection), spinflags);

         ret = -EPERM;
         if (type == CURSE_TARGETED) {
            if (((local_c->euid == 0) && (local_curse_perms & _SU_ACTIVE_PERM) && (foreign_
curse_perms & _SU_PASSIVE_PERM)) || \
                  (((local_c->euid == foreign_c->euid) || (local_c->euid == foreign_c->uid)
)                       && \
                   (local_curse_perms & _USR_ACTIVE_PERM) && (foreign_curse_perms & _USR_PA
SSIVE_PERM)))
               ret = 1;
         } else {
            if ((local_c->euid == 0) || (local_c->euid == foreign_c->euid) || (local_c->eui
d == foreign_c->uid))
               ret = 1;
         }

   }
   put_cred(foreign_c);
out_with_local:
   put_cred(local_c);
out:
   return ret;
}

/*This function takes a userspace string, and returns: 0 with the inode number in inode_num
ber, or error.*/
static int inode_from_user_path (char __user *path, unsigned long *inode_number)
{
   int ret = -ENOMEM;
   char *kernel_buffer;
   ssize_t len = (sizeof(path)+1);
   struct path tmp;
   umode_t in_mode;

   debug("Length is %d.\n", (int)len);
   if ((kernel_buffer = kzalloc(sizeof(char)*len, GFP_KERNEL)) == NULL)
      goto out;
   ret = -EFAULT;
   if (copy_from_user(kernel_buffer, path, len))
      goto out;
   debug("String is %s.\n", kernel_buffer);

   if ((ret = kern_path(/*transformed path*/ kernel_buffer, /*flags*/ LOOKUP_FOLLOW, &tmp))
)
      goto out;
   debug("kern_path return is %d.\n", ret);

   (*inode_number) = tmp.dentry->d_inode->i_ino;
   in_mode = tmp.dentry->d_inode->i_mode;

   debug("inode number is %lu and mode is %d\n", (*inode_number), (int)in_mode);
   if (!(in_mode & S_IXUGO)) {
      ret = -EPERM;
      debug("not executable\n");
   }

   path_put(&tmp);

out:
   return ret;
}

//=====Source syscall sub-functions.
static int syscurse_list_all (char __user *buf)
{
   int ret = -EINVAL;
   size_t length;
   //FIXME: I will add them for support, even if they are unused.
/*
   static size_t offset=0;

   if (len <= 0)
      goto out;
*/
   //length = sizeof(curse_full_list);
   length = sizeof(struct curse_list_entry)*max_curse_no;
// ret = ((length - offset) >= len) ? len : (length - offset);

   ret = 1;
// debug("My master you ask me to copy %u bytes, i shall do my best...\n", (unsigned int) l
ength);
   if (copy_to_user(buf, (const char *)&curse_full_list/*+offset*/, length)) {
      ret=-EFAULT;
      goto out;
   }
/*
   offset += ret;
   if (offset == length)
      offset=0;
*/
out:
   return ret;
}

static int syscurse_activate (int curse_no)
{
   int i, ret = -EPERM;

   i = curse_no;
   if ((ret = check_permissions(0, CURSE_SYSTEM)) != 1)
      goto out_ret;

   ret = 1;
   //Found a use for stub curse 0: activates the general curse system without activating an
y curse.
   if (bitmask_from_no(curse_no)) {                      //Activation of an existing curse,
 activates the system too.
      if (!(CURSE_FIELD(i, status) & ACTIVATED)) {
         CURSE_FIELD(i, status) |= ACTIVATED;
      } else {
         ret = -EINVAL;
         goto out_ret;
      }
   }
   if (!CURSE_SYSTEM_Q)                               //On invalid id, system activation.
      CURSE_SYSTEM_UP;

out_ret:
   return ret;
}

static int syscurse_deactivate (int curse_no)
{
   int i, ret = -EPERM;

   if ((ret = check_permissions(0, CURSE_SYSTEM)) != 1)
      goto out_ret;
   i = curse_no;

   ret = 1;
   if (bitmask_from_no(curse_no)) {                      //Targeted deactivation is normal.
      if (CURSE_FIELD(i, status) & ACTIVATED) {
         CURSE_FIELD(i, status) &= ~ACTIVATED;
      } else {
         ret = -EINVAL;
         goto out_ret;
      }
   } else if (/*!bitmask_from_no(curse_no) && */ CURSE_SYSTEM_Q)  //Invalid target deactiva
tes the system.
      CURSE_SYSTEM_DOWN;

   //TODO: Do we have to unhook (call close pointer) all the active curses here? :: No, we
simply deactivate. On activation, it will continue as was.

out_ret:
   return ret;
}

static int syscurse_check_curse_activity (int curse_no)
{
   int i, ret = -EINTR;

   if (!CURSE_SYSTEM_Q)
      goto out;

   i = curse_no;
   if (CURSE_FIELD(i, entry)->curse_id == 0xABADDE5C) {
      ret = -EINVAL;
      goto out;
   }
   if (CURSE_FIELD(i, status) & CASTED)
      ret = 1;
   else
      ret = 0;

out:
   return ret;
}

static int syscurse_check_tainted_process (int curse_no, pid_t target)
{
   int err = -EINVAL;
   uint64_t check_bit;
   unsigned long spinflags;
   struct task_struct *target_task;

   if (!(check_bit = bitmask_from_no(curse_no)) || (target <= 0))
      goto out;
   if (!CURSE_SYSTEM_Q)
      goto out;

   err = -ESRCH;
   rcu_read_lock();
   target_task = find_task_by_vpid(target);
   rcu_read_unlock();
   if (!target_task)
      goto out;

   err = -EINVAL;
   if (target <= 0)
      goto out;
   if ((err = check_permissions(target, CURSE_TARGETED)) != 1)
      goto out;
   err = 0;

   //Check if target has an active curse on it. :: FIXME: Move it to one-liner? Is it bette
r?
   spin_lock_irqsave(&((target_task->curse_data).protection), spinflags);
   if (target_task->curse_data.curse_field & check_bit)
      err = 1;
   else
      err = 0;
   spin_unlock_irqrestore(&((target_task->curse_data).protection), spinflags);

out:
   return err;
}
```

```
static int syscurse_ctrl (int curse_no, int ctrl, pid_t pid)
{
   int index, ret = -EINVAL;
   struct task_struct *target_task;
   struct task_curse_struct *cur_curse_field;
   unsigned long flags = 0;
   uint8_t ctrl_masks[] = {_USR_ACTIVE_PERM, _USR_PASSIVE_PERM, _SU_ACTIVE_PERM, _SU_PASSIV
E_PERM};
   _Bool set_clr;
   int com_index;

   index = curse_no;

   spin_lock_irqsave(&CURSE_FIELD(index, flag_lock), flags);
   ret = 1;
   switch (ctrl) {      /*Inherritance (on curse_list_pointer array)*/
   case INH_ON    :
      SET_INHER(index);
      break;
   case INH_OFF   :
      CLR_INHER(index);
      break;
   default:
      ret = -1;
   }
   spin_unlock_irqrestore(&CURSE_FIELD(index, flag_lock), flags);

   if (ret == 1)
      goto out;

   rcu_read_lock();
   target_task = find_task_by_vpid(pid);
   rcu_read_unlock();
   if (!target_task)
      goto out;
   cur_curse_field = &(target_task->curse_data);

   ret = -EINVAL;
   if (pid <= 0)
      goto out;
   if ((ret = check_permissions(pid, CURSE_REMOTE)) != 1) {
      goto out;
   }

   if ((ctrl >= USR_ACTIVE_PERM_ON) && (ctrl <= SU_PASSIVE_PERM_ON)) {
      set_clr=0;
      com_index = (ctrl - USR_ACTIVE_PERM_ON);
   } else if ((ctrl >= USR_ACTIVE_PERM_OFF) && (ctrl <= SU_PASSIVE_PERM_OFF)) {
      set_clr=1;
      com_index = (ctrl - USR_ACTIVE_PERM_OFF);
   } else {
      set_clr=2;
   }

   spin_lock_irqsave(&(cur_curse_field->protection), flags);
   switch (set_clr) {      /*Permissions (on task_curse_struct struct)*/
      case 0   :
         SET_PERM((*cur_curse_field), ctrl_masks[com_index]);
         break;
      case 1   :
         CLR_PERM((*cur_curse_field), ctrl_masks[com_index]);
         break;
      default  :
         ret = -EINVAL;
   }
   spin_unlock_irqrestore(&(cur_curse_field->protection), flags);

out:
   return ret;
}
```

```
static int syscurse_cast (int curse_no, pid_t target)
{
   int err = -EINVAL;
   unsigned long spinflags;
   struct task_struct *target_task;
   int new_index;
   uint64_t new_mask;

   if (!CURSE_SYSTEM_Q)
      goto out;


   err = -ESRCH;
   rcu_read_lock();
   target_task = find_task_by_vpid(target);
   rcu_read_unlock();
   if (!target_task)
      goto out;

   err = -EINVAL;
   if (target <= 0 )
      goto out;
   if ((err = check_permissions(target, CURSE_TARGETED)) != 1)
      goto out;

   err = -EINVAL;
   new_index = curse_no;
   new_mask = CURSE_FIELD(new_index, curse_bit);

   if ((!new_mask) || (!(CURSE_FIELD(new_index, status) & ACTIVATED)))
      goto out;

   spin_lock_irqsave(&((target_task->curse_data).protection), spinflags);
   if (!(target_task->curse_data.curse_field & new_mask)) {
      target_task->curse_data.curse_field |= new_mask;
      atomic_inc(&CURSE_FIELD(new_index, ref_count));
      if (GET_INHER(new_index))
         target_task->curse_data.inherritance |= new_mask;
      else
         target_task->curse_data.inherritance &= (~new_mask);
      CURSE_FIELD(new_index, status) |= CASTED;
      err = 1;
   }
   spin_unlock_irqrestore(&((target_task->curse_data).protection), spinflags);
   CURSE_FIELD(new_index, functions)->fun_init(target_task);   //Call init after cast.

out:
   return err;
}

static int syscurse_lift (int curse_no, pid_t target)
{
   int err = -EINVAL;
   unsigned long spinflags;
   struct task_struct *target_task;
   uint64_t curse_mask;
   int index;

   if (!CURSE_SYSTEM_Q)
      goto out;

   index = curse_no;
   err = -ESRCH;
   rcu_read_lock();
   target_task = find_task_by_vpid(target);
   rcu_read_unlock();
   if (!target_task)
      goto out;

   err = -EINVAL;
   if (target <= 0)
      goto out;
   if ((err = check_permissions(target, CURSE_TARGETED)) != 1)
      goto out;

   err = -EINVAL;
   if (!(curse_mask = CURSE_FIELD(index, curse_bit)))
      goto out;

   spin_lock_irqsave(&((target_task->curse_data).protection), spinflags);
   if (target_task->curse_data.curse_field & curse_mask) {
      target_task->curse_data.curse_field &= (˜curse_mask);    //Just to be safe (^= toggle
s, not clears).
      atomic_dec(&CURSE_FIELD(index, ref_count));
      target_task->curse_data.inherritance &= (˜curse_mask);
      if (atomic_read(&CURSE_FIELD(index, ref_count)) == 0)    //Revert curse status to ACT
IVATED if ref 0ed-out.  : Could be atomic_dec_and_set.
         CURSE_FIELD(index, status) &= ˜CASTED;
      err = 1;
   }
   spin_unlock_irqrestore(&((target_task->curse_data).protection), spinflags);

   CURSE_FIELD(index, functions)->fun_destroy(target_task); //Call destroy after lift.

out:
   return err;
}

static int syscurse_show_rules (void)
{
   return 0;
}

static int syscurse_add_rule (int curse, char __user *path)
{
   int ret = -EINVAL;
   unsigned long in_num;

   //Find inode
   //Check if executable
   if ((ret = inode_from_user_path(path, &in_num)))
      goto out;

   //Check permissions
   //Check if it is already in saved
   //Else do it

out:
   return ret;
}

static int syscurse_rem_rule (int curse, char *path)
{
   //Find inode
   //Check if it is in saved
   //Check permissions
   //Else do it
   return 0;
}

//=====Syscall kernel source.
/*This is the system call source base function.*/
SYSCALL_DEFINE5 (curse, unsigned int, curse_cmd, int, curse_no, pid_t, target, int, cur_ctr
l, char __user *, buf)     //asmlinkage long sys_curse(int curse_cmd, int curse_no, pid_t t
arget)
{
   long ret = -EINVAL;
   int cmd_norm = (int) curse_cmd;
   if ((curse_no < 0) || (curse_no >=max_curse_no))
      goto out;
```

```
// debug("Master, you gave me command %d with curse %d on pid %ld.\n", curse_cmd, curse_no,
 (long)target);

   //Do not even call if curse system is not active.
   switch (cmd_norm) {
   case LIST_ALL:
      ret = syscurse_list_all(buf);
      break;
   case CURSE_CTRL:
      ret = syscurse_ctrl(curse_no, cur_ctrl, target);
      break;
   case ACTIVATE:
      ret = syscurse_activate(curse_no);
      break;
   case DEACTIVATE:
      ret = syscurse_deactivate(curse_no);
      break;
   case CHECK_CURSE_ACTIVITY:
      ret = syscurse_check_curse_activity(curse_no);
      break;
   case CHECK_TAINTED_PROCESS:
      ret = syscurse_check_tainted_process(curse_no, target);
      break;
   case CAST:
      ret = syscurse_cast(curse_no, target);
      break;
   case LIFT:
      ret = syscurse_lift(curse_no, target);
      break;
   case GET_CURSE_NO:
      ret = max_curse_no;
      break;
   case SHOW_RULES:
      ret = syscurse_show_rules();
      break;
   case ADD_RULE:
      ret = syscurse_add_rule(curse_no, buf);
      break;
   case REM_RULE:
      ret = syscurse_rem_rule(curse_no, buf);
      break;
   case ILLEGAL_COMMAND:
   default:
      goto out;
   }

out:
   return ret;
}

#undef CURSE_SYSTEM
#undef CURSE_TARGETED
#undef CURSE_REMOTE

#else

SYSCALL_DEFINE5 (curse, unsigned int, curse_cmd, int, curse_no, pid_t, target, int, cur_ctr
l, char __user *, buf)
{
   return -ENOSYS;
}

#endif /* CONFIG_CURSES */
```

```
#include <linux/compiler.h>
#include <linux/types.h>      /*Sentinels prevent multiple inclusion.*/
#include <linux/sched.h>
#include <linux/spinlock.h>
#include <asm/atomic.h>

#include <curse/curse_list.h>
#include <curse/curse_types.h>
#include <curse/curse.h>      //Now it is only needed for the macros.

//=====Kernel functions.
#ifdef CONFIG_CURSES
//=====Global data.
/*Pointer to the implemented curse array (loaded at init of syscall).*/
struct syscurse *curse_list_pointer=(struct syscurse *)NULL;
/*Proc node pointer.*/
struct proc_dir_entry *dir_node=(struct proc_dir_entry *)NULL, *output_node=(struct proc_di
r_entry *)NULL;

/*Curse specific data - allocation interface.*/
#ifdef _CURSE_TASK_STRUCT_DEFINED

void *curse_create_alloc (struct task_struct *h, size_t desired_alloc_size, curse_id_t owne
r)
{
   void *ret;
   /*This function returns a pointer to a small amount of memory (ATOMIC).*/
   /*The memory can be accessed with the curse_get_mem method (in case we want it to be pro
cess specific),
    or with a static variable in the curse object (in case we want it to be common to all p
rocesses).*/

   if (!(ret = kmalloc(desired_alloc_size, GFP_ATOMIC))) {
      return NULL;
   } else {
      unsigned long tfs;
      struct curse_inside_data *tmp;

      tmp = (struct curse_inside_data *)kmalloc(sizeof(struct curse_inside_data), GFP_ATOMI
C);
      /*Create element 'offline'*/
      tmp->elem = ret;
      tmp->owner = owner;
      spin_lock_irqsave(&((h->curse_data).protection), tfs);
      /*Connect it to the list*/
      tmp->next = ((h->curse_data).use_by_interface).head;
      ((h->curse_data).use_by_interface).head = tmp;
      spin_unlock_irqrestore(&((h->curse_data).protection), tfs);
   }
   return ret;
}

void curse_free_alloc (struct task_struct *h, void *mem_to_free)
{
   /*Must be called with a pointer allocated with curse_get_alloc, else the system may get
destalibized.*/
   unsigned long tfs;
   struct task_curse_struct *hi;
   struct curse_inside_data *prev, *cur;

   hi = &(h->curse_data);
   spin_lock_irqsave(&((h->curse_data).protection), tfs);
   cur = (hi->use_by_interface).head;
   prev = cur;
   if (prev !=NULL) {
      while (cur != NULL) {
         /*Search for proper data pointer*/
         if (cur->elem == mem_to_free)
            break;
         prev = cur;
         cur = cur->next;
      }
      if (cur == NULL)
         goto out;
      /*Free data (and remove node too)*/
      kfree(cur->elem);
      if (((hi->use_by_interface).head) == cur)
         (hi->use_by_interface).head = (hi->use_by_interface).head->next;
      else
         prev->next = cur->next;
      kfree(cur);
   }
out:
   spin_unlock_irqrestore(&((h->curse_data).protection), tfs);
}

void *curse_get_mem (struct task_struct *h, curse_id_t cid)
{
   void *ret = NULL;
   unsigned long tfs;
   struct task_curse_struct *hi;
   struct curse_inside_data *rs;

   hi = &(h->curse_data);
   rs = (hi->use_by_interface).head;
   spin_lock_irqsave(&((h->curse_data).protection), tfs);
   /*If there are data*/
   while (rs != NULL) {
      /*Find the proper node*/
      if (rs->owner == cid) {
         ret = rs->elem;
         break;
      }
      rs = rs->next;
   }
   spin_unlock_irqrestore(&((h->curse_data).protection), tfs);
   return ret;
}

void curse_free_alloced_ll (struct task_struct *h)
{
   unsigned long tfs;
   struct curse_inside_data *c, *p;

   spin_lock_irqsave(&((h->curse_data).protection), tfs);
   p = ((h->curse_data).use_by_interface).head;
   /*Free all nodes*/
   if (p) {
      c = (p != NULL) ? (p->next) : NULL;
      while (p != NULL) {
         kfree(p->elem);
         kfree(p);
         p = c;
         if (c != NULL)
            c = c->next;
      }
      ((h->curse_data).use_by_interface).head = NULL;
   }
   spin_unlock_irqrestore(&((h->curse_data).protection), tfs);
}

#endif

//FIXME: Couldn’t we add a macro in curse_externals.h that changes id to mask during compil
ation? ::Possible conflicts with curse_init, that creates the masks.
static inline int index_from_curse_id (curse_id_t a_c_id)
{
   int i = 0;

   if (a_c_id == 0x00)
      goto out;
   for (i = 1; i < MAX_CURSE_NO; ++i)
```

```
      if ((curse_list_pointer[i].entry->curse_id) == a_c_id)
         goto out;

out:
   return i;
}

static int proc_curse_read (char *page, char **start, off_t off, int count, int *eof, void
*data)
{
   int i, line_len, ret = 0;
   /*We provided the data pointer during creation of read handler for our proc entry.*/
   struct syscurse *c_list = (struct syscurse *) data;

   if ((off > 0) || (data == NULL)) {  //Dunno; see here:   http://www.thehackademy.net/mad
chat/coding/procfs.txt  : We do not support reading continuation.
      (*eof) = 1;
      goto out;
   }

   //FIXME: Fix exaggeration: we have to predict that the next print will not cause an over
flow, so I am being overly cautious.
   line_len = sizeof(c_list[i].entry->curse_name) + sizeof(c_list[i].entry->curse_id);
   for (i = 0; ((i < max_curse_no) && ((ret + line_len) < count)); ++i)
      ret += scnprintf(&page[ret], count, "%s %llX\n", c_list[i].entry->curse_name, c_list[
i].entry->curse_id);
   (*start) = page;

out:
   return ret;
}

/*This is the injection wrapper, which must be in kernel space. This basically is an inline
 or define directive that checks if curses are activated and if the current process has a c
urse before calling the proper curse function.*/
void curse_k_wrapper (void)
{
   struct task_struct *cur;
   unsigned long flags;

   if (!CURSE_SYSTEM_Q)
      goto out;

   cur = current;
   //call the curse handler if there is a curse
   //if is used for opt, might integrate the handler here
   //ideas?

   if (cur->curse_data.curse_field) {
      int i = 1;
      uint64_t c_m = 0x0001;
      uint64_t c_f;
      uint64_t c_t;

      spin_lock_irqsave(&(cur->curse_data.protection), flags);
      c_f = cur->curse_data.curse_field;
      c_t = cur->curse_data.triggered;
      c_f &= c_t;

      //... This is where check and curse take place.
      while (c_f) {     //While the current is active, or there are remaining fields:
         if (c_f & c_m)
            fun_array[i].fun_inject(curse_list_pointer[i].curse_bit);
         c_f >>= 1;
         ++i;
      }
      cur->curse_data.triggered = 0x00;
      spin_unlock_irqrestore(&(cur->curse_data.protection), flags);
   }

out:
   return;
}

/*This function initializes all needed resources (only) once, during system init.*/
void curse_init (void)
{
   int j;
   curse_id_t t;

   //1. Initialize curse lookup table.
   curse_list_pointer = (struct syscurse *)kzalloc((MAX_CURSE_NO + 1) * sizeof(struct syscu
rse), GFP_KERNEL);
   if (curse_list_pointer == NULL) {
      printk(KERN_CRIT "CRITICAL: Curse system was not able to allocate memory. The system
will probably crash later.");
      goto out;
   }
   for (j = 1, t = 0x01; j < MAX_CURSE_NO; ++j, t <<= 1) {
      curse_list_pointer[j].entry = (struct curse_list_entry *)&curse_full_list[j];
      curse_list_pointer[j].curse_bit = t;
      atomic_set(&(curse_list_pointer[j].ref_count), 0);
      curse_list_pointer[j].var_flags = _INHER_MASK;
      SET_INHER(j);
      curse_list_pointer[j].status = IMPLEMENTED;
      spin_lock_init(&(curse_list_pointer[j].flag_lock));
      curse_list_pointer[j].functions = &fun_array[j];
   }
   curse_list_pointer[0].status = INVALID_CURSE;
   curse_list_pointer[0].curse_bit = 0x0;
   atomic_set(&(curse_list_pointer[0].ref_count), 0);
   curse_list_pointer[0].entry = (struct curse_list_entry *)&curse_full_list[0];
   spin_lock_init(&(curse_list_pointer[0].flag_lock));
   curse_list_pointer[0].functions = &fun_array[0];

   //2. Initialize active status boolean. :: Could default on an initial status here (based
 on build options).
   CURSE_SYSTEM_DOWN;

   //3. Populate entries in /proc filesystem.
   if (!(dir_node = proc_mkdir(PROC_DIR_NAME, NULL)))
      goto out;
   if (!(output_node = create_proc_read_entry(PROC_OUT_NODE_NAME, (S_IRUSR | S_IRGRP | S_IR
OTH), dir_node, proc_curse_read, curse_list_pointer)))
      goto out_dirred;

   //FIXME: Is there anything else to be done here?

   goto out;
//out_nodded:
   remove_proc_entry(PROC_OUT_NODE_NAME, dir_node);
out_dirred:
   remove_proc_entry(PROC_DIR_NAME, NULL);
out:
   return;     //Stub: there might be others below.
}

/*This function is inserted in the places of the kernel source code that act as triggers fo
r each curse, and inserts a trigger indicator in task struct of each task.*/
//FIXME: May have to swap out with define directive. Also, remove excessive overhead.
void curse_trigger (_Bool defer_action, curse_id_t cid)
{
   struct task_curse_struct *cur_struct;
   unsigned long spinf;
   int index;

// debug("Trigger on %lld\n", cid);
   index = index_from_curse_id(cid);

   cur_struct = &(current->curse_data);

   if (!unlikely(defer_action)) {
      uint64_t proc_active;

      spin_lock_irqsave(&((current->curse_data).protection), spinf); //Check if curse is  a
ctive.
      proc_active = curse_list_pointer[index].curse_bit;
      spin_unlock_irqrestore(&((current->curse_data).protection), spinf);
      if (!(proc_active &= current->curse_data.curse_field))
         return;
      (curse_list_pointer[index].functions)->fun_inject(curse_list_pointer[index].curse_bit
);
   } else {
      spin_lock_irqsave(&(cur_struct->protection), spinf);
      cur_struct->triggered |= (curse_list_pointer[index].curse_bit);
      spin_unlock_irqrestore(&(cur_struct->protection), spinf);
   }

}

void curse_init_actions (struct task_struct *p)
{
   int i = 1;
   uint64_t c_m = 0x0001, c_f = p->curse_data.curse_field;

   //REMOVED: Have to check if system is active before acting. Active bits don’t get toggle
d when system inactive.
   while (c_f) {     //While the current is active, or there are remaining fields:
      //debug("INIT ON FORK: This process has curses %llX.\n", c_f);
      if ((c_f & c_m) && (curse_list_pointer[i].status & CASTED)) {
         fun_array[i].fun_init(p);
         //debug("The before ref value is %d.\n", atomic_read(&(curse_list_pointer[i].ref_c
ount)));
         atomic_inc(&(curse_list_pointer[i].ref_count));
         //debug("The after ref value is %d.\n", atomic_read(&(curse_list_pointer[i].ref_co
unt)));
         curse_list_pointer[i].status |= CASTED;
      }
      c_f >>= 1;
      ++i;
   }
   //...
}

void curse_destroy_actions (struct task_struct *p)
{
   int i = 1;
   uint64_t c_m = 0x0001, c_f = p->curse_data.curse_field;

   while (c_f) {     //While the current is active, or there are remaining fields:
      if ((c_f & c_m) && (curse_list_pointer[i].status & (ACTIVATED | CASTED))) {
         //debug("DESTROY ON EXIT: This process has curse with index %d.\n", i);
         fun_array[i].fun_destroy(p);
         //debug("The before ref value is %d.\n", atomic_read(&(curse_list_pointer[i].ref_c
ount)));
         atomic_dec(&(curse_list_pointer[i].ref_count));
         //debug("The after ref value is %d.\n", atomic_read(&(curse_list_pointer[i].ref_co
unt)));
         if (atomic_read(&(curse_list_pointer[i].ref_count)) == 0)
            curse_list_pointer[i].status &= ~CASTED;
      }
      c_f >>= 1;
      ++i;
   }
   if (p->curse_data.curse_field)
      curse_free_alloced_ll(p);
   //...
}

/*Define dummies here, for the case when the curses system is not inserted in the kernel co
de.*/
/* Not needed for all of them. Maybe just trigger. Everything else should be protected with
 the CONFIG_CURSES guard.*/
```

```
#else

void curse_trigger (_Bool cond, curse_id_t _)
{
   return;
}

#endif   /* CONFIG_CURSES */
```

```
/*
 * This library is the main library for the curse system call.
 * It is to be included by both userspace and kernel programs, so we take care to define th
e public interface properly.
 *
 * Since we want it to be located in the same directory with the curse source file,
 *  it will be included by it in double quotes,
 *  but the userspace inclusion is to be done in the normal fashion.
 */

#ifndef _SYSCURSE_H
#define _SYSCURSE_H

#ifndef __KERNEL__         /*Inclusion of uint64_t on userspace.*/
#include <stdint.h>
#endif
#include <curse/curse_types.h>

/*Curse system call interface.*/
enum curse_command   {  LIST_ALL=0, CURSE_CTRL,
                  ACTIVATE, DEACTIVATE,
                  CHECK_CURSE_ACTIVITY,
                  CHECK_TAINTED_PROCESS,
                  CAST, LIFT, GET_CURSE_NO,
                  SHOW_RULES,
                  ADD_RULE, REM_RULE,
                  ILLEGAL_COMMAND
               };

/*Curse control commands.*/
enum curse_control   {  INH_ON=0, INH_OFF,
                  USR_ACTIVE_PERM_ON, USR_PASSIVE_PERM_ON,
                  SU_ACTIVE_PERM_ON, SU_PASSIVE_PERM_ON,
                  //GRP_PERM_ON, GRP_PERM_OFF,
                  USR_ACTIVE_PERM_OFF, USR_PASSIVE_PERM_OFF,
                  SU_ACTIVE_PERM_OFF, SU_PASSIVE_PERM_OFF
               };

/*Lists every possible status for a curse (for userspace portability).*/
enum curse_status {IMPLEMENTED=0x00, ACTIVATED=0x01, CASTED=0x02, INVALID_CURSE=0x04};

/*Procfs entry names.*/
#define PROC_DIR_NAME "curse"
#define PROC_OUT_NODE_NAME "curse_list"

//TODO: Cleanup and check comments. Also move around things between kernel and userspace. S
ee header.
#ifdef __KERNEL__

/*Kernel specific libraries.*/
#include <linux/proc_fs.h>    /*struct proc_dir_entry*/
#include <linux/types.h>      /*pid_t, uin64_t*/
#include <asm/atomic.h>       /*atomic_t*/

/*Structure describing a curse (and its status).*/
struct syscurse {
   struct curse_list_entry *entry;     //Not sure if it should be just struct or pointer, b
ecause problems may arise during copy to userspace.
   atomic_t ref_count;              //Count of how many active deployments exist for this c
urse.
   uint64_t curse_bit;              //Corresponding bitfield for the current curse.
   spinlock_t flag_lock;
   uint8_t var_flags;               //Flags field.
   enum curse_status status;        //Activation status for this curse.
   struct curse_fun_element *functions;
};

/*Pointer to the implemented curse array (loaded at init of syscall).*/
extern struct syscurse *curse_list_pointer;
/*Proc node pointer.*/
extern struct proc_dir_entry *dir_node, *output_node;
```

```
/*Inheritance specific macros (curse-specific inheritance is inserted in var_flags field of
 syscurse struct.*/
#define _INHER_MASK  0x20
#define GET_INHER(_index) (((curse_list_pointer[_index]).var_flags) & (_INHER_MASK))
#define SET_INHER(_index) (((curse_list_pointer[_index]).var_flags) |= (_INHER_MASK))
#define CLR_INHER(_index) (((curse_list_pointer[_index]).var_flags) &= ~(_INHER_MASK))

/*Bitmasks to use for setting and checking the permissions field in struct tast_curse_struc
t.*/
/*Active permissions denote a capability to cast/lift = Passive permissions denote a capabi
lity to have a curse cast upon us*/
#define _USR_ACTIVE_PERM   0x01
#define _USR_PASSIVE_PERM  0x02
//#define _GRP_ACTIVE_PERM 0x04
//#define _GRP_PASSIVE_PERM   0x08
#define _SU_ACTIVE_PERM    0x10
#define _SU_PASSIVE_PERM   0x20
/*Permission specific macros (first argument is a task_curse_struct variable, and the secon
d a permission mask).*/
#define GET_PERM(el, perm_mask) (((el).permissions) & (perm_mask))
#define SET_PERM(el, perm_mask) (((el).permissions) |= (perm_mask))
#define CLR_PERM(el, perm_mask) (((el).permissions) &= ~(perm_mask))

/*This macro gives encapsulated access to the curse system general status.*/
#define CURSE_SYSTEM_Q (atomic_read(&(curse_list_pointer[0].ref_count)))
#define CURSE_SYSTEM_DOWN atomic_set(&(curse_list_pointer[0].ref_count), 0)
#define CURSE_SYSTEM_UP atomic_set(&(curse_list_pointer[0].ref_count), 1)

//DEBUG macro for development.
#ifdef CONFIG_CURSE_DEBUG
#define debug(fmt,arg...)     printk(KERN_INFO "%s: " fmt, __func__ , ##arg)
#else
#define debug(fmt,arg...)     do { } while(0)
#endif

#ifndef curse_struct
#define curse_struct(target) ({                                \
   unsigned long int __sfl;                                 \
   struct task_curse_struct ret_data;                          \
   spin_lock_irqsave(&((target->curse_data).protection),__sfl);      \
   ret_data = (target->curse_data);                          \
   spin_unlock_irqrestore(&((target->curse_data).protection),__sfl); \
   ret_data;                                              \
   })
#endif

#endif   /* __KERNEL__ */

#endif /* _SYSCURSE_H */
```

```
/*
 * This file is part of the interface between the curses mechanism
 * and the curses implementation.
 * Every curse available must be registered here.
 * TODO: Maybe we could add a description field in each curse.
 *
 */
#ifdef CONFIG_CURSES

#ifndef _CURSE_LIST_LIB
#define _CURSE_LIST_LIB

#ifdef __KERNEL__

#include <linux/types.h>
#include <curse/curse_types.h>

/*Maximum number of curses (1 is the lower limit).*/

/*[ADD] The individual curse header includes.*/
#include <curse/stub_curse.h>
#include <curse/no_curse.h>
#include <curse/no_fs_cache.h>
#include <curse/random_oops.h>
#include <curse/poison.h>
#include <curse/no_exit.h>
#include <curse/test_curse.h>

#ifndef MAX_CURSE_NO
#define MAX_CURSE_NO 1
#endif

/*[ADD] The system curse listing.*/
struct __attribute__((packed)) curse_list_entry curse_full_list[] = {
   {  "system", 0x00000000 },

#ifdef CONFIG_NO_CURSE
   {  "no_curse", 0xBEA7CE5C  },
#endif
#ifdef CONFIG_NO_FS_CACHE
   {  "no_fs_cache", 0x00000002  },
#endif
#ifdef CONFIG_RANDOM_OOPS
   {  "random_oops", 0xDEFEC8ED  },
#endif
#ifdef CONFIG_POISON
   {  "poison", 0xDEADBEEF },
#endif
#ifdef CONFIG_NO_EXIT
   {  "no_exit", 0xCAFECAFE   },
#endif
#ifdef CONFIG_TEST
   {  "test", 0x01010101   },
#endif

   {  "sentinel", 0xABADDE5C  }  /*Curse table sentinel. Every entry after this will be ign
ored.*/
};

#undef MAX_CURSE_NO
#define MAX_CURSE_NO (((sizeof curse_full_list)/(sizeof (struct curse_list_entry)))-1)

/* External linking for number of curses. Kernelspace only */
const int max_curse_no = (((sizeof (curse_full_list))/(sizeof (struct curse_list_entry)))-1
);

/*[ADD] The system call function pointer array.*/
struct curse_fun_element fun_array[] = {
   {  stub_init, stub_destroy, stub_inject   }, /* Maybe a stub maybe not, depends on how w
e handle 0 :: It is a stub handling curse system activation */
#ifdef CONFIG_NO_CURSE
   {  no_curse_init, no_curse_destroy, no_curse_inject   },
#endif
#ifdef CONFIG_NO_FS_CACHE
   {  no_fs_cache_init, no_fs_cache_destroy, no_fs_cache_inject   },
#endif
#ifdef CONFIG_RANDOM_OOPS
   {  stub_init, stub_destroy, random_oops_inject  },
#endif
#ifdef CONFIG_POISON
   {  poison_init, poison_destroy, poison_inject   },
#endif
#ifdef CONFIG_NO_EXIT
   {  stub_init, stub_destroy, no_exit_inject   },
#endif
#ifdef CONFIG_TEST
   {  test_init, test_destroy, test_inject   },
#endif

   {  stub_init, stub_destroy, stub_inject   } /* you have made a grave mistake (sentinel s
peaking) */
};

#endif   /* __KERNEL__ */

#endif /* _CURSE_LIST_LIB */
#endif /* CONFIG_CURSES */
```

```
/*
 * This library is to be included in the sched/fs/exec/fork sources,
 * so as not to include unnecessary definitions and libraries.
 */

#ifndef _CURSE_TYPES_LIB
#define _CURSE_TYPES_LIB

typedef uint64_t curse_id_t;

/* Kernel-specific structures. */
#ifdef __KERNEL__

#include <linux/types.h>
#include <linux/spinlock.h>

#ifndef _CURSE_TASK_STRUCT_DEFINED
#define _CURSE_TASK_STRUCT_DEFINED

/*Curse specific data (linked list head element).*/
struct curse_inside_data {
   void *elem;
   curse_id_t owner;
   struct curse_inside_data *next;
};

/*Struct to-be injected in task_struct to let us keep tabs on processes.*/
struct task_curse_struct {
   spinlock_t protection;     //Because it is included in sched.h (and no semaphores are we
lcome there:))
   uint64_t triggered;
   uint64_t curse_field;
   uint64_t inherritance;     //Bitwise association of this field’s bits and the previous o
ne’s.
   uint8_t permissions;

   uint32_t no_fs_cache_counter;
   uint32_t poison_counter;

   struct curse_specific_data {
      struct curse_inside_data *head;
   } use_by_interface;
};

#endif   /* _CURSE_TASK_STRUCT_DEFINED */

/*System call function pointer structure.*/
struct curse_fun_element {
   void (*fun_init) (struct task_struct * );
   void (*fun_destroy) (struct task_struct * );
   void (*fun_inject) (uint64_t);
};

#endif   /* __KERNEL__ */

/*Maximum size for a curse name.*/
#define CURSE_MAX_NAME_SIZE 24

/*Curse entry structure for logistic purposes.*/
struct __attribute__((packed)) curse_list_entry {
    char curse_name[CURSE_MAX_NAME_SIZE];
    curse_id_t curse_id;
};

#endif   /* _CURSE_TYPES_LIB */
```

```
#ifdef CONFIG_CURSES
#ifdef CONFIG_NO_FS_CACHE
#include <linux/fadvise.h>
#include <linux/fdtable.h>
#include <linux/syscalls.h>
#include <linux/spinlock.h>

#include <curse/no_fs_cache.h>
#include <curse/curse.h>
#include <curse/curse_externals.h>

void no_fs_cache_init (struct task_struct *target)
{
   /*
    * we don't need to, if its greater than MAX_NO_FS_COUNT
    * it will be re initialized automagically :)
    */
   uint32_t *counter = NULL;

   counter = curse_create_alloc(target, sizeof(uint32_t), 0x00000002);
   if (counter != NULL) {
      *counter = 0;
   }

   return;
}

void no_fs_cache_destroy (struct task_struct *target)
{
   uint32_t *counter = NULL;

   curse_trigger(0, 0x00000002);
   counter = curse_get_mem(target, 0x00000002);
   curse_free_alloc(target, counter);
   counter = NULL;

   return;
}

static inline void clear_cache_loop (int lim) {
   int n;
   for (n = 0; n <= lim; ++n) {
      if (fcheck(n)) {
         sys_fadvise64_64(n, 0, 0, POSIX_FADV_DONTNEED);
         //debug("%ld's got sth up %d\n", (long)current->pid, n);
      }
   }
}

void no_fs_cache_inject (uint64_t mask)
{
   /* http://linux.die.net/man/2/fadvise */

   struct fdtable *fdt;
   struct files_struct *open_files;
   uint32_t *counter;
   unsigned long irqflags;
   spinlock_t *curse_lock = NULL;
   *curse_lock = curse_struct(current).protection;

   counter = curse_get_mem(current, 0x00000002);
   if (*counter > MAX_NO_FS_COUNT) {
      rcu_read_lock();
//    preempt_disable();      //FIXME: Possible fix?

      open_files = get_files_struct(current);
      fdt = files_fdtable(open_files);

      clear_cache_loop(fdt->max_fds);

//    preempt_enable();    //FIXME: Possible fix?
```

```
      rcu_read_unlock();
      put_files_struct(open_files);

      spin_lock_irqsave(curse_lock, irqflags);
      *counter = 0;
      spin_unlock_irqrestore(curse_lock, irqflags);
   } else {
      spin_lock_irqsave(curse_lock, irqflags);
      ++(*counter);
      spin_unlock_irqrestore(curse_lock, irqflags);
   }

   return;
}

#endif   /* CONFIG_NO_FS_CACHE */
#endif   /* CONFIG_CURSES */
```