OSM
G assignment 3

Tobias Hallundbæk Petersen (xtv657)
Ola Rønning (vdl761)
Nikolaj Høyer (ctl533)

March 3rd, 2014

Contents

1 Task 1: A Thread-Safe Stack **2**

2 Task 2: Userland Semaphores for Buenos **2**

1 Task 1: A Thread-Safe Stack

2 Task 2: Userland Semaphores for Buenos

We have added the following cases to proc/syscall.c:

```
    case SYSCALL_SEM_OPEN:
      V0 = (int) syscall_sem_open((char *)A1, A2);
      break;
    case SYSCALL_SEM_PROCURE:
      V0 = syscall_sem_p((void *)A1);
      break;
    case SYSCALL_SEM_VACATE:
      V0 = syscall_sem_v((void *)A1);
      break;
    case SYSCALL_SEM_DESTROY:
      V0 = syscall_sem_destroy((void *)A1);
      break;
```

The function calls stated above resides in their own file, an addition to proc, namely proc/usr_semaphore.c and proc/usr_semaphore.h. Hence, in syscall.c we add the line #include "proc/usr_semaphore.h". We have also added a new syscall to proc/syscall.h: #define SYSCALL_SEM_DESTROY 0x303 to cover the case of this specific syscall.

In syscall.h we define the new user land semaphore structure, which contains of a pointer to a kernel semaphore and a name of the userland semaphore, along with function definitions.

```
#include "kernel/semaphore.h"
#define MAX_NAME_LEN 20

typedef struct {
  semaphore_t* sem;
  char name[MAX_NAME_LEN];
} usr_sem_t;

void usr_semaphore_init(void);
usr_sem_t* syscall_sem_open(char const* name, int value);
int syscall_sem_p(usr_sem_t* handle);
int syscall_sem_v(usr_sem_t* handle);
int syscall_sem_destroy(usr_sem_t* handle);
```

The real functionality is implemented in proc/syscall.c.

In addition to the functions specified in the assignment text, we have added an init function and added the following lines to init/main.c:

```
kwrite("Initializing userland semaphores\n");
usr_semaphore_init();
```

The init function simply runs through our userland semaphore table, which we have defined in `proc/usr_semaphore.c`, setting all its user semaphores to `NULL`. This table has a size that is a constant fraction of the kernel semaphores, we have set it to `CONFIG_MAX_SEMAPHORES / 8`. Since the userland semaphores are actually based on kernel semaphores (creating a new userland semaphore will actually create a kernel semaphore), we could get in a situation where the user could fully populate the semaphore table, making the kernel routines unable to create any semaphores at all.

The open function takes care of either retrieving an existing user semaphore (value < 0) or creating a new user semaphore (value $>= 0$). Since we're accessing the user semaphore table, we need to disable interrupts and set a spinlock. Most of the functionality revolves around accessing the user semaphore table, comparing the `name` input variable to the names stored in the `usr_sem_t` structures stored in the semaphore table. The inline comments should describe what happens, and when.

The `sem_p` and `sem_v` function simply calls the underlying kernel semaphore functionality - since this in itself provides interruption disabling and spinklocks, we don't need to apply that here.

The destroy function simply runs through the user semaphore table, invalidating all the semaphores and freeing the user semaphore table slots.

The code for `proc/usr_semaphores.c` is shown below:

```
#include "kernel/semaphore.h"
#include "kernel/config.h"
#include "kernel/interrupt.h"
#include "kernel/kmalloc.h"
#include "proc/usr_semaphore.h"
#include "lib/libc.h"
#define MAX_USR_SEMAPHORES CONFIG_MAX_SEMAPHORES / 8

<<<<<<< HEAD
/* Configure a separate userland semaphore table of the
 * same size as the kernel semaphore table.
 * Hence, this will never be fully populated as long as
 * there are active kernel semaphores. */
=======
/* Configure a separate userland semaphore table
 * - this will never be fully populated as long as there
 * are active kernel semaphores. */
```

```c
>>>>>>> 00d379deccf9e83be39ddeb5aebe451b5f281e36
static usr_sem_t usr_semaphore_table[MAX_USR_SEMAPHORES];

static spinlock_t usr_semaphore_table_slock;

void usr_semaphore_init(void)
{
  int i;
  interrupt_status_t intr_status;
  intr_status = _interrupt_disable();
  spinlock_reset(&usr_semaphore_table_slock);
  spinlock_acquire(&usr_semaphore_table_slock);
<<<<<<< HEAD

  // Initiale all userland semaphores to NULL
=======
>>>>>>> 00d379deccf9e83be39ddeb5aebe451b5f281e36
  for(i = 0; i < MAX_USR_SEMAPHORES; i++) {
    usr_semaphore_table[i].sem = NULL;
  }

  spinlock_release(&usr_semaphore_table_slock);
  _interrupt_set_state(intr_status);
}

usr_sem_t* syscall_sem_open(char const *name, int value)
{
  usr_sem_t *usr_sem = NULL;
  interrupt_status_t intr_status;
  int i;

  if (strlen(name) > MAX_NAME_LEN) {
    return NULL;
  }

  intr_status = _interrupt_disable();
  spinlock_acquire(&usr_semaphore_table_slock);

  // Get an existing userland semaphore (value < 0)
  if (value < 0) {
<<<<<<< HEAD
    for (i = 0; i < MAX_USR_SEMAPHORESS; i++) {
=======
    for (i = 0; i < MAX_USR_SEMAPHORES; i++) {
>>>>>>> 00d379deccf9e83be39ddeb5aebe451b5f281e36
      if(stringcmp(usr_semaphore_table[i].name, name) == 0) {
        usr_sem = &usr_semaphore_table[i];
        spinlock_release(&usr_semaphore_table_slock);
        _interrupt_set_state(intr_status);
        return usr_sem;
      }
    }
    // No semaphore with given name exists
    spinlock_release(&usr_semaphore_table_slock);
    _interrupt_set_state(intr_status);
    return NULL;
  }
  // Create new userland semaphore (value >= 0)
  else {
    int sem_id = MAX_USR_SEMAPHORES;
    for (i = 0; i < MAX_USR_SEMAPHORES; i++) {
<<<<<<< HEAD
      // Semaphore already exists
=======
>>>>>>> 00d379deccf9e83be39ddeb5aebe451b5f281e36
      if(stringcmp(usr_semaphore_table[i].name, name) == 0) {
        spinlock_release(&usr_semaphore_table_slock);
        _interrupt_set_state(intr_status);
        return NULL;
      }
      // Find an available spot in the userland semaphore table
      if (i < sem_id && usr_semaphore_table[i].sem == NULL) {
        sem_id = i;
      }
    }
    /* If there is no more space in the userland semaphore table,
     * return an error. This should never happen, since the actual
     * kernel semaphore table would be full before this could occur */
    if (sem_id == MAX_USR_SEMAPHORES) {
      return NULL;
    }
    // Create the actual userland semaphore
    usr_semaphore_table[sem_id].sem = semaphore_create(value);
    stringcopy(usr_semaphore_table[sem_id].name, name, MAX_NAME_LEN);
    usr_sem = &usr_semaphore_table[sem_id];

    spinlock_release(&usr_semaphore_table_slock);
    _interrupt_set_state(intr_status);

    return usr_sem;
  }
  // Something went wrong
  spinlock_release(&usr_semaphore_table_slock);
  _interrupt_set_state(intr_status);
  return NULL;
}

int syscall_sem_p(usr_sem_t* handle)
{
  semaphore_P(handle->sem);
  return 0;
}

int syscall_sem_v(usr_sem_t* handle)
{
  semaphore_V(handle->sem);
  return 0;
}

int syscall_sem_destroy(usr_sem_t* handle)
```

```
{
  int i;
  interrupt_status_t intr_status;

  intr_status = _interrupt_disable();
  spinlock_acquire(&usr_semaphore_table_slock);

  /* Look through the user semaphore table for
   * a usr_sem_t address matching the handle */
  for(i = 0; i < MAX_USR_SEMAPHORES; i++) {
    if(&usr_semaphore_table[i] == handle) {
      /* Invalidate the kernel semaphore and set the entry
       * in the userland semaphore table to empty */
      semaphore_destroy(handle->sem);
      usr_semaphore_table[i].sem = NULL;
      spinlock_release(&usr_semaphore_table_slock);
      _interrupt_set_state(intr_status);
      return 0;
    }
  }
  // If no match is found, return an error
  spinlock_release(&usr_semaphore_table_slock);
  _interrupt_set_state(intr_status);
  return 1;
}
```