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

1.1 a: Implementing the stack

When implementing a threadsafe stack, what needed to be changed from the non-threadsafe stack, was to implement mutual exclusion locks when access to the stack was needed. Furthermore as the size of the stack needed to be dynamic, the doubly linked list from G-assignment 1 was here implemented, and used as a singly linked list. This ensures that the stack can grow and shrink depending on the data.

A design choice was to return 0 if a pop was executed, and the stack was empty, this makes it easier to check whether the stack is empty in task 1.b, furthermore we avoid using conditions. This could of course also be implemented with conditions that waited for something to be pushed and then popped it afterwards, for our matrix multiplication we would then have to use `pthread_cancel` on the threads once we assured that the stack was empty.

We added the functionality in the doubly linked list, to peek at the head or tail, this is used when a `stack_top` is issued.

The header file for the stack is mostly similar to the one handed out, the only difference is that we use the doubly linked list for data.

```
#ifndef STACK_H
#define STACK_H

#include "dllist.h"

typedef struct stack_t {
  dlist* data;
  int top;
} stack_t;

/* Initialise a freshly allocated stack.  Must be called before using
   any of the other stack functions. */
void stack_init(stack_t*);

/* Returns true if the stack is empty. */
int stack_empty(stack_t*);

/* Return the top element of the stack.  Undefined behaviour if the
   stack is empty. */
void* stack_top(stack_t*);

/* Remove the top element of the stack and return it.  Undefined
   behaviour if the stack is empty. */
void* stack_pop(stack_t*);
```

```
/* Push an element to the top of the stack.  Returns 0 if possible,
   any other value if there was an error (for example, if the stack is
   full or no memory could be allocated). */
int stack_push(stack_t*, void*);


#endif
```

The stack is implemented as follows:

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include "stack.h"
#include "dllist.h"

pthread_mutex_t          mutex = PTHREAD_MUTEX_INITIALIZER;

void stack_init(stack_t* stack) {
  pthread_mutex_init(&mutex, NULL);
  dlist *list = malloc(sizeof(dlist));
  stack->data = list;
  stack->top = -1;
}

int stack_empty(stack_t* stack) {
  pthread_mutex_lock(&mutex);
  int retval = (stack->top == -1);
  pthread_mutex_unlock(&mutex);
  return retval;
}

void* stack_top(stack_t* stack) {
  pthread_mutex_lock(&mutex);
  if (stack->top == -1) {
    pthread_mutex_unlock(&mutex);
    return (void*) 0;
  }
  void* retval = peek(stack->data,0);
  pthread_mutex_unlock(&mutex);
  return retval;
}

void* stack_pop(stack_t* stack) {
  pthread_mutex_lock(&mutex);
  if (stack->top == -1) {
    pthread_mutex_unlock(&mutex);
    return (void*) 0;
  }
  stack->top--;
  void* retval = extract(stack->data,0);
  pthread_mutex_unlock(&mutex);
  return retval;
```

```
}

int stack_push(stack_t* stack, void* data) {
  pthread_mutex_lock(&mutex);
  stack->top++;
  insert(stack->data,data,0);
  pthread_mutex_unlock(&mutex);
  return 1;
}
```

1.2 b: Fixed thread matrix multiplication

The changes needed to the handed out matrix multiplication, were subtle as well. A funtion which we call `worker` pops tasks, which represent rows, from the stack and uses the provided function `rowmult` on the tasks, this continues until `stack_top` returns 0, at which point the function will return.
We wanted the second argument given to determine how many threads would be used, thus this needed to be implemented.
Lastly for fun and education we added a timer, showing us how much time it takes to complete the matrix multiplication.
The code for this task is implemented as follows:

```
/* Matrix multiplication row-wise in separate threads
 *
 * Compile with:
 *   gcc -Wall -Wextra -std=c99 -pedantic -o matmult matmult.c stack.c dllist.↩
    c -lpthreads
 *
 * Usage:
 *   ./PThread_matmult [SIZE (default = 100)] [NUMBER OF THREADS (default = 4)↩
    ]
 */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include "stack.h"
#include <sys/time.h>

#define DEFAULT_SIZE 100
#define DEFAULT_THREADS_SIZE 4
/* Data structure to drive a thread */
typedef struct {
  double *row_a;
  double *matrix_b;
  int a_length, b_columns;
  double *row_result;
} ttask_t;
```

```
stack_t stack;

/* Zero the result row of doubles in memory, compute dot product of row A with
 * all columns of B, return.
 */
void* rowmult(void *arg) {
  int i, j;
  ttask_t *t;

  t = (ttask_t*) arg;

  for (j = 0; j < t->b_columns; j++) {
    t->row_result[j] = 0.0;
  }
  for (i = 0; i < t->a_length; i++) {
    for (j = 0; j < t->b_columns; j++) {
      t->row_result[j] += t->row_a[i] * t->matrix_b[i * (t->b_columns) + j];
    }
  }
  return NULL;
}

void* worker(void *param) {
  param = param;
  ttask_t *t;
  while (1) {
    t = stack_pop(&stack);
    if (!t) {
      break;
    }
    rowmult(t);
  }
  return NULL;
}

/* Output a matrix of dimensions size x size. */
void output_square_matrix(double *m, int size) {
  int i, j;

  for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++) {
      printf("%3.2f  ", m[i * size + j]);
    }
    putchar('\n');
  }
}

/* Main program: Fill two matrices (randomly or with particular patterns; for
 * instance diagonals to "mirror" a matrix or some blocks of it), then create
 * one thread per row to multiply. Join all threads, then output the matrix (↩
    or
 * just declare success).
 *
 * Parameters: matrix size
```

```
 */
int main(int argc, char* argv[]) {
  int i, j;
  int size,no_of_threads;

  double *matrices, *a, *b, *r;

  pthread_t *threads;
  ttask_t *tasks, *t;
  char* num_end;

  /* Initialize stack */
  stack_init(&stack);
  /* Find out which size to compute with. */
  if (argc > 1) {
    size = strtol(argv[1], &num_end, 10);
    if (num_end[0] != '\0') {
      fprintf(stderr, "argument not a number: %s\n", argv[1]);
      exit(EXIT_FAILURE);
    }
    if (size < 1) {
      fprintf(stderr, "size negative or zero: %d\n", size);
      exit(EXIT_FAILURE);
    }
  }
  else {
    size = DEFAULT_SIZE;
  }
  if (argc > 2) {
    no_of_threads = strtol(argv[2], &num_end, 10);
    if (num_end[0] != '\0') {
      fprintf(stderr, "argument not a number: %s\n", argv[2]);
      exit(EXIT_FAILURE);
    }
    if (no_of_threads < 1) {
      fprintf(stderr, "threads negative or zero: %d\n", no_of_threads);
      exit(EXIT_FAILURE);
    }
  } else {
    no_of_threads = DEFAULT_THREADS_SIZE;
  }
  printf("Number of threads: %d\n",no_of_threads);
  printf("Multiplying random square matrices of size %d x %d\n",
         size, size);
 /* Allocate memory for both input matrices and the result matrix. */
  matrices = (double*) malloc(sizeof(double) * size * size * 3);
  if (matrices == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
  }
  a = matrices;
  b = matrices + size * size;
  r = matrices + size * size * 2;
  /* Allocate memory for threads and thread structures. */
  threads = (pthread_t*) malloc(no_of_threads * sizeof(pthread_t));
```

```
  if (threads == NULL) {
    perror("malloc");
    free(matrices);
    exit(EXIT_FAILURE);
  }
  tasks = (ttask_t*) malloc(size * sizeof(ttask_t));
  if (tasks == NULL) {
    perror("malloc");
    free(matrices);
    free(threads);
    exit(EXIT_FAILURE);
  }
   /* Fill matrices with values. a is random, b is diagonal. */
  srand(time(NULL)); /* A very bad RNG, but standard */

  for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++) {
      a[i * size + j] = (double) rand();
      b[i * size + j] = 0.0;
    }
    b[i * size + (size - i - 1)] = 1.0; /* second diagonal */
  }

  /* If size is small, output matrices. */
  if (size < 20) {
    printf("\nMatrix A:\n");
    output_square_matrix(a, size);
    printf("\nMatrix B (diagonal):\n");
    output_square_matrix(b, size);
  }

  struct timeval start, end;
  long mtime, secs, usecs;
  gettimeofday(&start, NULL);
  /* Add the threads to the stack. */
  for (i = 0; i < size; i++) {
    t = &tasks[i];
    t->row_a = a + i * size;
    t->matrix_b = b;
    t->a_length = size;
    t->b_columns = size;
    t->row_result = r + i * size;
    stack_push(&stack, t);
  }
  /* Start the threads. */
  for (i = 0; i < no_of_threads; i++) {
    if (pthread_create(&threads[i], NULL, worker, NULL) != 0) {
      perror("pthread_create");
      free(matrices);
      free(threads);
      free(tasks);
      exit(EXIT_FAILURE);
    }
  }
  /* Wait for the threads to finish. */
```

```
  for(i = 0; i < no_of_threads; i++) {
    pthread_join(threads[i], NULL);
    putchar('.');
  }
  putchar('\n');
  /* If size is small, output result matrix. */
  if (size < 20) {
    printf("\nResult:\n");
    output_square_matrix(r, size);
  }
  gettimeofday(&end, NULL);
  secs  = end.tv_sec  - start.tv_sec;
  usecs = end.tv_usec - start.tv_usec;
  mtime = ((secs) * 1000 + usecs/1000.0) + 0.5;
  printf("\nFinished in %lu milliseconds!\n", mtime);

  free(matrices);
  free(threads);
  free(tasks);
  free(stack.data);
  exit(EXIT_SUCCESS);
}
```

For testing we tried to compute different sized of matrices, with 1, 2, 3 and 4 threads, while keeping track of how many cores were used by the process, which coincided with how many threads we required.

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

/* 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. */

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);

  // Initiale all userland semaphores to NULL
  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) {
    for (i = 0; i < MAX_USR_SEMAPHORES; i++) {
      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++) {
      // Semaphore already exists
      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;
}
```

For testing we used the handed out program `tests/barrier.c` and the two programs that are run by it `tests/prog0.c` and `tests/prog1.c`. We ran `barrier` and it ran as desired, which concludes our testing.