Operating systems and multiprogramming

G-assignment 3

Alexander Worm Olsen - bdj816

Allan Martin Nielsen - jcl187

Troels Thompsen - qvw203

March 3 2014

Department of Computer Science
University of Copenhagen

A thread-safe stack

We were asked to implement a thread-safe stack, e.g. it should allow concurrent access by different threads safely. The stack implementation is to be found in *stack.c* and the hand-out interface is found in *stack.h*.

In the interface handed-out we changed the type of stacks data from an array to a double-linked-list, where we used our implementation from G1. This allowed our stack to be dynamic and therefore hold data of large matrices.

The implementation of the functions allowed by our interface are straight forward. For our pop and push we used a mutex lock to ensure safe concurrent access, and then we popped and pushed using the extract and insert from the double-linked-list. These insert and extract are all done to the head of the stack, which gives a first in-last out implementation.

Please note that in the initialisation of our stack (stack_init()) we push a pointer to a '0' on the stack, this is done due to the fact, that our dlist implementation of our G1 does not work in the case where only 2 elements is inserted and then both extracted. The implementation and 'hack' is shown in listing 1.

Listing 1: The stack_init implementation

```
void stack_init(stack_ty* stack) {
    pthread_mutex_init(&lock, NULL);
    stack->size = -1;
    void* data = 0;
    insert((&stack->datalist),data,0);
}
```

Matrix multiplication using threads

In order to test the implementation of our thread-safe stack we were to implement a matrix multiplication program using a fixed number of threads. We've used the handed-out *Pthread_matmult.c* and modified it to use a fixed number of threads.

We haven't changed much in the hand-out, in summary we added a fixed number of threads, e.g. 10. The tasks, each row, is pushed onto our thread-safe stack and then we've changed the Pthread creations to use a for-loop creating exactly the fixed number of threads and then each of these threads run a new helper function called do_task with a popped task from the stack. This helper function makes sure the threads keep running until all the work is done and then the threads are joined. Below in listing 2 we've included our helper function, which is the essential code snippet for our threads.

Listing 2: The helper function do_task

```
void* do_task(void *task) {
    while(task) {
        rowmult(task);
        if(stack_top(&stack) != 0){
        task = stack_pop(&stack);
    }
    }
    return NULL;
}
```

Userland semaphores

We were asked to implement the functionality of userland semaphores to Buenos. We've done this mainly in the proc folder with the two files *semaphore.c* and *semaphore.h*. We made the choice of implementing a table to hold all the userland semaphores where they could be mapped to kernel semaphores. Therefore we've implemented a usr_sem_init where we initialise the table.

Our implementation of syscall_sem_open uses a lot of nesting and various conditions. This is due to the fact, that the name must be of valid length, the value can either mean a creation or a look-up. If the semaphore is to be created this is done by the code shown in listing 3.

Listing 3: Creation of new userland semaphores

```
    for(i = 0; i < MAX_SEMAPHORES; i++) {
        if(usr_sem_table[i].SId == -1) {
            usr_sem_table[i].kernel_sem = semaphore_create(value);
            usr_sem_table[i].SId = 0;
            stringcopy(usr_sem_table[i].sem_name,name,MAX_NAME_SIZE);
            return &(usr_sem_table[i]);
        }
```

The implementation of vacate and procure is straight forward, first we find the semaphore in the table, and then we call the kernel semaphore functions.

In the implementation of destroy we've tried to handle the various problems that might take place with a spinlock.

Testing

In order to test the implemented userland semaphores we used the hand-out *barrier.c, prog1.c, prog0.c*. We excluded a test file for the syscall_sem_destroy, this is

mainly due to lack of time, but we tried to avoid synchronisation problems, although we havn't taken care of other threads already blocking it in the sleep queue because anything we tried just returned errors.