Operating Systems
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: 320202
Date: 2015-02-25
Deadline: 2015-03-04

Problem Sheet #2

Problem 2.1: *pthread programming* (3+2+1 = 6 points)

Take a look at the C code shown below. Assume that all system and library calls succeed at runtime. (All error handling code has been omitted for brevity.)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#include <unistd.h>

#define N 5

static long int a[N];

typedef struct {
    pthread_t thread;
    int       tid;
    long int  value;
} thread_state_t;

static void* weird(void *arg)
{
    thread_state_t *ts = (thread_state_t *) arg;
    int me = ts->tid;
    int ne = (me > 0) ? me - 1 : N - 1;

    while (1) {
        if (a[ne] == -1) {
            a[me] = a[ne];
            break;
        } else if (a[ne] == ts->value) {
            a[me] = -1;
            printf("\n%d: The lucky number is: %ld\n", me, a[ne]);
            break;
        } else if (a[ne] > a[me]) {
            a[me] = a[ne];
        } else {
            /* do nothing */
        }
    }

    return NULL;
}

int main(int argc, char *argv[])
{
    int i;
    thread_state_t ts[N];

    srandom(getpid());          /* initialize the random number generator */
    for (i = 0; i < N; i++) {
        ts[i].value = a[i] = random();  /* a random non-negative number */
        ts[i].tid = i;                  /* an id in the range [0..N-1] */
        printf("%d: %12ld\n", ts[i].tid, ts[i].value);
```

```
    }

    for (i = 0; i < N; i++) {
        (void) pthread_create(&ts[i].thread, NULL, weird, &ts[i]);
    }

    for (i = 0; i < N; i++) {
        (void) pthread_join(ts[i].thread, NULL);
    }

    return EXIT_SUCCESS;
}
```

a) Explain what the program is doing. What is the lucky number? which thread is going to print the lucky number? Hint: Try an example with N=3 to see what the algorithm does.

b) The threads operate on shared data and hence there might be synchronization problems. What is the shared data? How can the potential synchronization problem be fixed? Insert statements as you see fit to fix any race conditions. clear).

c) Is the program always going to produce the expected result?

Solution:

a) The array `a` contains N random numbers and N threads are created. The array is treated as a cyclic buffer. Each thread goes into a loop comparing the value of its left neighbor (`ne`). If the neighbor's value is -1, then the thread's value in the array is set to -1 and the thread breaks out of the loop. If the neighbor's value equals the value of the thread (`ts->value`), then the thread found a lucky number and sets the thread's value in the array to -1. If the neighbor's value is larger than the thread's own value in the array, then the value is overwritten with the neighbor's value.

   As a consequence, the large value propagates through the array `a` overwriting smaller values. One the thread "owning" the largest value detects that his neighbor's value is the same as his value (which means his value has propagated through the whole array), the thread prints the lucky number (which is the largest number in the array) and subsequently sets its value in the array to the special marker -1 to let the other threads know that the largest value was found.

b) The array `a` is shared data that is read and written by the concurrent threads. Introducing a mutex to enforce mutual exclusion in the while loop fixes this problem.

c) The program assumes that the random numbers in the array `a` are all unique. If same random number shows up more than once in the array, the algorithm has a certain chance to terminate reporting the wrong lucky number.

Problem 2.2: *pthreads and stacks* (4 points)

The pthread library allows programs to control the size of the stacks used by threads. Write a program that uses `pthread_attr_getstacksize()` to show the default stack size and that subsequently uses `pthread_attr_setstacksize()` to set the stack size to the minimum stack size given by `PTHREAD_STACK_MIN` (you may need to include `limits.h`).

Your program should subsequently create a thread with the minimum stack size that executes a recursive function. What happens once the thread runs out of stack space? Explain why the system reacts in the way it does. (Hint: You may want to read about guard areas controlled by the `guardsize` thread attribute.)

Solution:

```
/*
 * This program creates a thread with the minimum stack size and lets
```

```
 * the thread execute an infinite recursion. The program fails with a
 * segmentation fault. Why does it fail? The pthread library allocates
 * guard pages that are neither readable or writable. This is
 * controlled by the guardsize thread attribute. By setting the
 * guardsize to 0, threads may happily continue outsize their stack
 * areas. Not really recommended...
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <pthread.h>

#define check_error(err, msg) \
    do { \
        if (err) { errno = err; perror(msg); exit(EXIT_FAILURE); } \
    } while (0);

static int level = 0;

static void *
recurse(void *last)
{
    char data[960];
    char p;

    printf("%-8d %-16p\n", level++, &p);

    return recurse(data);
}

int main()
{
    pthread_t thread_id;
    pthread_attr_t attr;
    size_t stack_size;
    int s;

    s = pthread_attr_init(&attr);
    check_error(s, "pthread_attr_init");

    s = pthread_attr_getstacksize(&attr, &stack_size);
    check_error(s, "pthread_attr_getstacksize");

    printf("default stack size is %zu bytes\n", stack_size);

    s = pthread_attr_setstacksize(&attr, PTHREAD_STACK_MIN);
    check_error(s, "pthread_attr_setstacksize");

    s = pthread_attr_getstacksize(&attr, &stack_size);
    check_error(s, "pthread_attr_getstacksize");

    printf("minimum stack size is %zu bytes\n", stack_size);

    s = pthread_create(&thread_id, &attr, recurse, NULL);
    check_error(s, "pthread_create");

    s = pthread_join(thread_id, NULL);
    check_error(s, "pthread_join");

    return EXIT_SUCCESS;
}
```