Operating Systems and Networks
International University Bremen
Dr. Jürgen Schönwälder

Course: 320202
Date: 2004-02-29
Deadline: 2004-03-05

Problem Sheet #3

Problem 3.1: *bridge crossing synchronization problem* (8 points)

An old bridge which crosses a river from east to west. Since it is a small bridge, cars can only go in one direction at a time and no more than 3 cars are allowed on the bridge.

- Cars arriving at the bridge while the bridge is empty will immediately cross the bridge.
- Cars arriving at the bridge while there are cars on the bridge traveling in the wrong direction will wait until the bridge becomes empty.
- Cars arriving at the bridge while there are less than three cars already on the bridge in the same direction will immediately cross the bridge.
- Cars arriving at the bridge while there are three cars on the bridge traveling in the same direction will wait until the first car leaves the bridge and then immediately cross the bridge.

Write a program using the `pthread` library which will use multiple threads to simulate cars arriving at the bridge. Every thread basically executes the following function

```
static void
drive(bridge_t *bridge, int direction)
{
    arrive(bridge, direction);
    cross(bridge);
    leave(bridge);
}
```

where `bridge` is a shared data structure which represents the bridge including any required synchronization variables. Your implementation should use pthread mutexes (`pthread_mutex_t`) and condition variables (`pthread_cond_t`). Your implementation of `cross()` should call `sleep(1)` in order to force context switches. Explain your solution of the synchronization problem by commenting your source code.

Note: It is not required to come up with a starvation free solution.

Solution:

```
/*
 * bridge/bridge.c --
 *
 * An old bridge which crosses a river from east to west. Since it is
 * a small bridge, cars can only go in one direction at a time and no
 * more than 3 cars are allowed on the bridge.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define BRIDGE_DIRECTION_EAST 0x01
#define BRIDGE_DIRECTION_WEST 0x02
```

```
typedef struct bridge {
    int cars;
    int direction;
    pthread_mutex_t mutex;
    pthread_cond_t  empty;
} bridge_t;

static bridge_t shared_bridge = {
    .cars = 0,
    .direction = BRIDGE_DIRECTION_WEST,
    .mutex = PTHREAD_MUTEX_INITIALIZER,
    .empty = PTHREAD_COND_INITIALIZER,
};

/*
 * First wait until there are either no cars on the bridge of less
 * than two cars traveling into our direction. If there are no cars,
 * we can just define the new direction.
 */

static void
arrive(bridge_t *bridge, int direction)
{
    pthread_mutex_lock(&bridge->mutex);
    while (bridge->cars > 0 &&
           (bridge->cars > 2 || bridge->direction != direction)) {
        pthread_cond_wait(&bridge->empty, &bridge->mutex);
    }
    if (bridge->cars == 0) {
        bridge->direction = direction;
    }
    bridge->cars++;
    pthread_mutex_unlock(&bridge->mutex);
}

/*
 * Crossing the bridge is easy - just some delay. Note that without
 * calling sleep(), there is a good chance that threads will just run
 * until completion, which is kind of boring.
 */

static void
cross(bridge_t *bridge)
{
    fprintf(stderr, "crossing %d ... cars = %d \n",
            bridge->direction, bridge->cars);
    sleep(1);
}

/*
 * Leave the bridge by waking up other cars who might be waiting
 * in front of the bridge.
 */

static void
leave(bridge_t *bridge)
{
    pthread_mutex_lock(&bridge->mutex);
    bridge->cars--;
    pthread_cond_signal(&bridge->empty);
    pthread_mutex_unlock(&bridge->mutex);
}
```

```
/*
 * All cars implement the same behavior.
 */

static void
drive(bridge_t *bridge, int direction)
{
    arrive(bridge, direction);
    cross(bridge);
    leave(bridge);
}

/*
 * Entry point for threads going east.
 */

static void*
east(void *data)
{
    drive((bridge_t *) data, BRIDGE_DIRECTION_EAST);
    return NULL;
}

/*
 * Entry point for threads going west.
 */

static void*
west(void *data)
{
    drive((bridge_t *) data, BRIDGE_DIRECTION_WEST);
    return NULL;
}

/*
 * Run nw threads to go west and ne threads to go east. This function
 * could be smarter to introduce some random delays and to alternate
 * better between west-bound and east-bound cars.
 */

static int
run(int nw, int ne)
{
    int i, n = nw + ne;
    pthread_t thread[n];

    for (i = 0; i < n; i++) {
        if (pthread_create(&thread[i], NULL,
                           i < nw ? east : west, &shared_bridge)) {
            fprintf(stderr, "thread creation failed\n");
            return EXIT_FAILURE;
        }
    }

    for (i = 0; i < n; i++) {
        if (thread[i]) pthread_join(thread[i], NULL);
    }

    return EXIT_SUCCESS;
}

int
```

```
main(int argc, char **argv)
{
    int c, nw = 1, ne = 1;
    const char *usage = "Usage: bridge [-e east] [-w west] [-h]\n";

    while ((c = getopt(argc, argv, "e:w:h")) >= 0) {
        switch (c) {
        case 'e':
            if ((ne = atoi(optarg)) <= 0) {
                fprintf(stderr, "number of cars going east must be > 0\n");
                exit(EXIT_FAILURE);
            }
            break;
        case 'w':
            if ((nw = atoi(optarg)) <= 0) {
                fprintf(stderr, "number of cars going west must be > 0\n");
                exit(EXIT_FAILURE);
            }
            break;
        case 'h':
            printf(usage);
            exit(EXIT_SUCCESS);
        }
    }

    return run(nw, ne);
}
```

Problem 3.2: *semaphore queuing strategies* (2 points)

The definition of semaphores so far has been silent about the strategy used to dequeue waiting processes. Consider the two strategies first-in-first-out (FIFO) and last-in-first-out (LIFO). Which strategy is better? Provide a convincing explanation.

Solution:

- The FIFO dequeuing strategy guarantees that a queued process will be dequeued once all processes in front of the process have been dequeued. In other words, the time spend in the queue only depends on the number of processes already in the queue while entering the queue.
- The LIFO dequeuing strategy causes a process to stay in the queue until all processes entering the queue after the process have been dequeued again. The time spend in the queue therefore depends on the number of processes queued in the future. This means that starvation is possible if a constant stream of processes enters (and leaves) the queue.