

## Exercise Sheet 3

### Assignment 3.1 Lockfree vs. locked programming

The purpose of the following exercises is to get acquainted with the `pthread` library, locks and lockfree algorithms.

1. Implement the bumper `alloc`.
2. demonstrate, that allocating memory concurrently produces inconsistent results
3. repair your implementation with
  - lockfree instructions
  - a pthread semaphore

compare your 3 implementations considering correctness and performance!

### Suggested Solution 3.1

1. We get races in `bumper.c`
2. According wait-free implementation in `bumperwaitfree.c`
3. According implementation with a semaphore/mutex in `bumpersemaphore.c`

### Assignment 3.2 Lockfree Algorithms

Given the following data structures:

```
typedef struct node {
    int val;
    struct node* next;
} node;
typedef struct{
    node* top;
} stack;
```

and the following code:

```
void push(stack* s, int i){
    node* newtop = malloc(sizeof(node));
    newtop->val=i;
    newtop->next = s->top;
    s->top = newtop;
}
```

1. Replace `push` with your own function `push_lockfree`, which is made threadsafe, without the use of locks. Instead you may use **lockfree** instructions, e.g.

```
_compare_and_swap (type *ptr, type oldval, type newval)
```

*atomically stores **newval** into **ptr**, if **\*ptr** evaluates to **oldval** and returns **\*ptr** before eventually overwriting it, for **type** being an arbitrary type*

2. Provide an `int* pop_lockfree(stack* s)` function, that pops stack `s` threadsafe lockfree, returning null if the stack is empty and a pointer to the integer at the top of the stack otherwise.

### Suggested Solution 3.2

1. 

```
void pushlockfree(stack* s,int i){
    node* newtop = malloc(sizeof(node));
    newtop->val=i;
    while (1){
        newtop->next = s->top;
        if (__sync_val_compare_and_swap(&s->top,newtop->next,newtop) == newtop->next)
            return;
    }
}
```
2. 

```
int* pop_lockfree(stack * s){
    node* top;
    node* next;
    do{
        if ((top = s->top)==0) return 0;
        next= top->next;
        if(__sync_val_compare_and_swap(&s->top,top,next)==top)
            break;
    }while(1);
    int * ret = malloc(sizeof(int));
    *ret = top->val;
    free(top);
    return ret;
}
```

### Assignment 3.3 Parallel Programming – Monitors

Find the functions in the `pthread` library that provide you with semaphores, monitors, and condition variables. Start your research with `pthread_mutex_init`.

Consider the following code, implementing basic functionality for the doubly linked list:

1. Upgrade the queue to be threadsafe, using a single mutex. Implement the `ForAll` method and use it in `main` to cause a deadlock.

2. Implement the queue using a monitor and show that the deadlock goes away.

```
// gcc Dqueue_pure.c -o dqueue
#include <stdlib.h> // malloc
typedef struct QNode {
    int val;
    struct QNode* left;
    struct QNode* right;
} QNode;
typedef struct {
    struct QNode* left;
    struct QNode* right;
} DQueue;

void PushLeft(DQueue* q, int val) {
    QNode *qn = (QNode *)malloc(sizeof(QNode));
    qn->val = val;
    QNode* leftSentinel = q->left;
    QNode* oldLeftNode = leftSentinel->right;
    qn->left = leftSentinel;
    qn->right = oldLeftNode;
    leftSentinel->right = qn;
    oldLeftNode->left = qn;
}

int PopRight(DQueue* q) {
    QNode* oldRightNode;
    QNode* leftSentinel = q->left;
    QNode* rightSentinel = q->right;
    oldRightNode = rightSentinel->left;
    if(oldRightNode == leftSentinel)
        return -1;
    QNode* newRightNode = oldRightNode->left;
    newRightNode->right = rightSentinel;
    rightSentinel->left = newRightNode;
    int ret = oldRightNode->val;
    free(oldRightNode);
    return ret;
}

void Forall(DQueue* q, void* data, void (*callback)(void*, int)) {
    // Executes callback on all items of this list
}

int main() {
    // init
    DQueue *q = (DQueue*)malloc(sizeof(DQueue));
    QNode* sentinel = (QNode*)malloc(sizeof(QNode));
    q->right = sentinel;
    q->left = sentinel;
    sentinel->right = sentinel;
```

```
sentinel->left = sentinel;  
  
// fill initial load  
  
// ... code to prepare a deadlock  
  
// Forall  
  
// ... produce a deadlock  
  
// end all  
}
```

### Suggested Solution 3.3

1. see `DQueue.c`
2. see `DQueueMonitor.c`