PIOTR WOŚ

SHARED MEMORY LINKED LIST

https://github.com/ptr97/shared-memory-linked-list

PROJECT DESCRIPTION

- **“Complex objects with pointers (say linked list) in shared memory”**
- For simplicity let’s say that:
  - we have only one producer and only one consumer
  - producer and consumer both work sequentially - firstly producer adds something to the list, then consumer reads from there

PROJECT DESCRIPTION

- I will show you solution which uses arithmetic on offsets from the beginning of shared memory
- And if we still have some time left I will tell you about hardest one which should work (and almost works ;)) with use of arithmetic on pointers

PROJECT DESCRIPTION

- Shared memory
  - mmap
  - shmget, shmat
  - I chosen mmap because of easy, natural access to it (you can easily look inside the file)
- Why structures like linked list aren’t easy to store in shared memory?

THE CHALLANGE

- One and only one challange is the fact that virtual address of shared memory may be and almost always is different each time.

- Is it all?
- Yea… pointer arithmetic…
- And… many ideas for solution.

THE SOLUTION(S)

```cpp
#include <iostream>
#include "SharedMemory.h"
#include "List.h"


int main()
{
    const uint shmBlockSize = 4096 * 2;

    List<int> list = List<int>::createListInShm("database.db", shmBlockSize);
    list.add(10);
    list.add(20);
    list.add(30);
    list.add(40);
    list.print();
    std::cout << "list.exists(20) == " << list.exists(20) << std::endl;
    list.remove(20);
    list.remove(10);
    list.print();
    shmBlock::freeShm(shmBlockSize);

    std::cout << std::endl << "Now we will read linked list from memory" << std::endl;
    List<int> listFromMem = List<int>::readListFromMemory("database.db");
    listFromMem.print();
    shmBlock::freeShm(shmBlockSize);

    return 0;
}
```

```
Memory block of size 8192 allocated at: 0x109533000
List capacity: 682
freeOffset = 0
freeOffset = 1
freeOffset = 2
freeOffset = 3
Current: 40
Current: 30
Current: 20
Current: 10
list.exists(20) == 1
removing node with value 20
removing node with value 10
Current: 40
Current: 30
Memory block has been released...

Now we will read linked list from memory
Memory block of size 8192 read at: 0x109533000
List capacity: 682
Current: 40
Current: 30
Memory block has been released...
```

THE SOLUTION(S)

```
Memory block of size 8192 allocated at: 0x102f51000
List capacity: 682
freeOffset = 0
freeOffset = 1
freeOffset = 2
freeOffset = 3
Current: 40
Current: 30
Current: 20
Current: 10
Memory block has been released...
```

```
Memory block of size 8192 read at: 0x10dd85000
List capacity: 682
Current: 40
Current: 30
Current: 20
Current: 10
Memory block has been released...
```

SHARED MEMORY LINKED LIST

THE SOLUTION(S)

```h
#ifndef __SHARED_MEMORY_H__
#define __SHARED_MEMORY_H__

#include <iostream>
#include <cstring>
#include <cstdlib>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <sys/mman.h>
#include <unistd.h>


class shmBlock
{
public:
  static char * startPtr;

  static void allocateMemory(const char * key, uint size);
  static uint readFromMemory(const char * key);
  static void freeShm(uint size);
};

#endif
```

```cpp
#include "SharedMemory.h"

char * shmBlock::startPtr;

void shmBlock::allocateMemory(const char * key, uint size) {
  const int fd = open(key, O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  if( fd < 0 ) {
    std::cout << "Open DB error" << std::endl;
    exit(-1);
  }

  const int status = ftruncate(fd, static_cast<off_t>(size));
  if(status != 0) {
    std::cout << "ftrucate error" << std::endl;
    exit(-1);
  }

  shmBlock::startPtr = (char *) mmap(0, size, PROT_WRITE, MAP_SHARED, fd, 0);
  if(shmBlock::startPtr == MAP_FAILED) {
    std::cout << "mmap failed" << std::endl;
    exit(-1);
  }

  memset(shmBlock::startPtr, '\0', static_cast<off_t>(size));
  std::cout << "Memory block of size " << size << " allocated at: " << (void *) shmBlock
}
```

```cpp
void shmBlock::freeShm(uint size) {
  if(munmap(shmBlock::startPtr, static_cast<off_t>(size))) {
    std::cout << "unmap error" << std::endl;
    exit(-1);
  }
  std::cout << "Memory block has been released..." << std::endl;
}
```

```cpp
uint shmBlock::readFromMemory(const char * key) {
  const int fd = open(key, O_RDONLY, S_IRUSR | S_IWUSR);
  if(fd < 0) {
    std::cout << "Open DB error" << std::endl;
    exit(-1);
  }

  struct stat file_statistics;
  fstat(fd, &file_statistics);

  shmBlock::startPtr = (char *) mmap(0, file_statistics.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
  if(shmBlock::startPtr == MAP_FAILED) {
    std::cout << "mmap failed" << std::endl;
    exit(-1);
  }
  std::cout << "Memory block of size " << file_statistics.st_size << " read at: " << (void *) shmBlock::st
  return file_statistics.st_size;
}
```

THE SOLUTION(S)

```
16  template <typename T>
17  class List {
18    template <typename U>
19    struct Node {
20      Node(const U u, int offset) {
21        this->value = u;
22        this->nextOffset = offset;
23      }
24
25      U value;
26      int nextOffset = -1;
27      bool saved = true;
28    };
29
30    struct Meta {
31      unsigned int head;
32    };
33
```

```
34  public:
35    static List<T> createListInShm(const char * shmName, uint shmBlockSize) {
36      shmBlock::allocateMemory(shmName, shmBlockSize);
37      return List(true, shmBlockSize);
38    }
39
40    static List<T> readListFromMemory(const char * shmName) {
41      uint shmSize = shmBlock::readFromMemory(shmName);
42      return List(false, shmSize);
43    }
```

```
102 private:
103   List(bool newList, uint shmBlockSize) {
104     struct List<T>::Meta & meta = getMeta();
105     setCapacity(shmBlockSize);
106     if (newList) {
107       meta.head = -1;
108       for(int i = 0; i < maxSize; ++i) {
109         getNode(i).saved = false;
110       }
111     }
112   }
```

THE SOLUTION(S)

```
114   Node<T> & getNode(int offset) {
115     return *reinterpret_cast<Node<T>*>(shmBlock::startPtr + sizeof(Meta) + sizeof(Node<T>) * offset);
116   }
117
118   Meta & getMeta() {
119     return *reinterpret_cast<Meta *>(shmBlock::startPtr);
120   }
121
122   int findFreeOffset() {
123     int next_offset = 0;
124     while (getNode(next_offset).saved != false) {
125       if (next_offset > maxSize) {
126         std::cout << "ERROR: Out of shm memory" << std::endl;
127         exit(-1);
128       }
129       ++next_offset;
130     }
131     return next_offset;
132   }
133
134   void setCapacity(uint shmBlockSize) {
135     maxSize = (shmBlockSize - sizeof(Meta)) / sizeof(Node<T>);
136     std::cout << "List capacity: " << maxSize << std::endl;
137   }
138
139   int maxSize;
140 };
```

THE SOLUTION(S)

```
45  void add(T item){
46    int freeOffset = findFreeOffset();
47
48    std::cout << "freeOffset = " << freeOffset << std::endl;
49    Meta & meta = getMeta();
50    Node<T> & newNode = getNode(freeOffset);
51    newNode.value = item;
52    newNode.nextOffset = meta.head;
53    newNode.saved = true;
54
55    meta.head = freeOffset;
56  }
57
58  bool exists(T item) {
59    Meta & meta = getMeta();
60    int current = meta.head;
61    while (current != -1) {
62      const Node<T> & currentNode = getNode(current);
63      if (currentNode.value == item) {
64        return true;
65      } else {
66        current = currentNode.nextOffset;
67      }
68    }
69    return false;
70  }
```

```
72  bool remove(T item) {
73    Meta & meta = getMeta();
74    int current = meta.head;
75    int last = meta.head;
76    while (current != -1) {
77      Node<T> & currentNode = getNode(current);
78      if (currentNode.value == item) {
79        std::cout << "removing node with value " << currentNode.value <<
80        currentNode.saved = false;
81        Node<T> & lastNode = getNode(last);
82        lastNode.nextOffset = currentNode.nextOffset;
83        return true;
84      } else {
85        last = current;
86        current = currentNode.nextOffset;
87      }
88    }
89    return false;
90  }
91
92  void print() {
93    Meta & meta = getMeta();
94    int current = meta.head;
95    while (current != -1) {
96      const Node<T> & currentNode = getNode(current);
97      std::cout << "Current: " << currentNode.value << std::endl;
98      current = currentNode.nextOffset;
99    }
100 }
```

RESULT

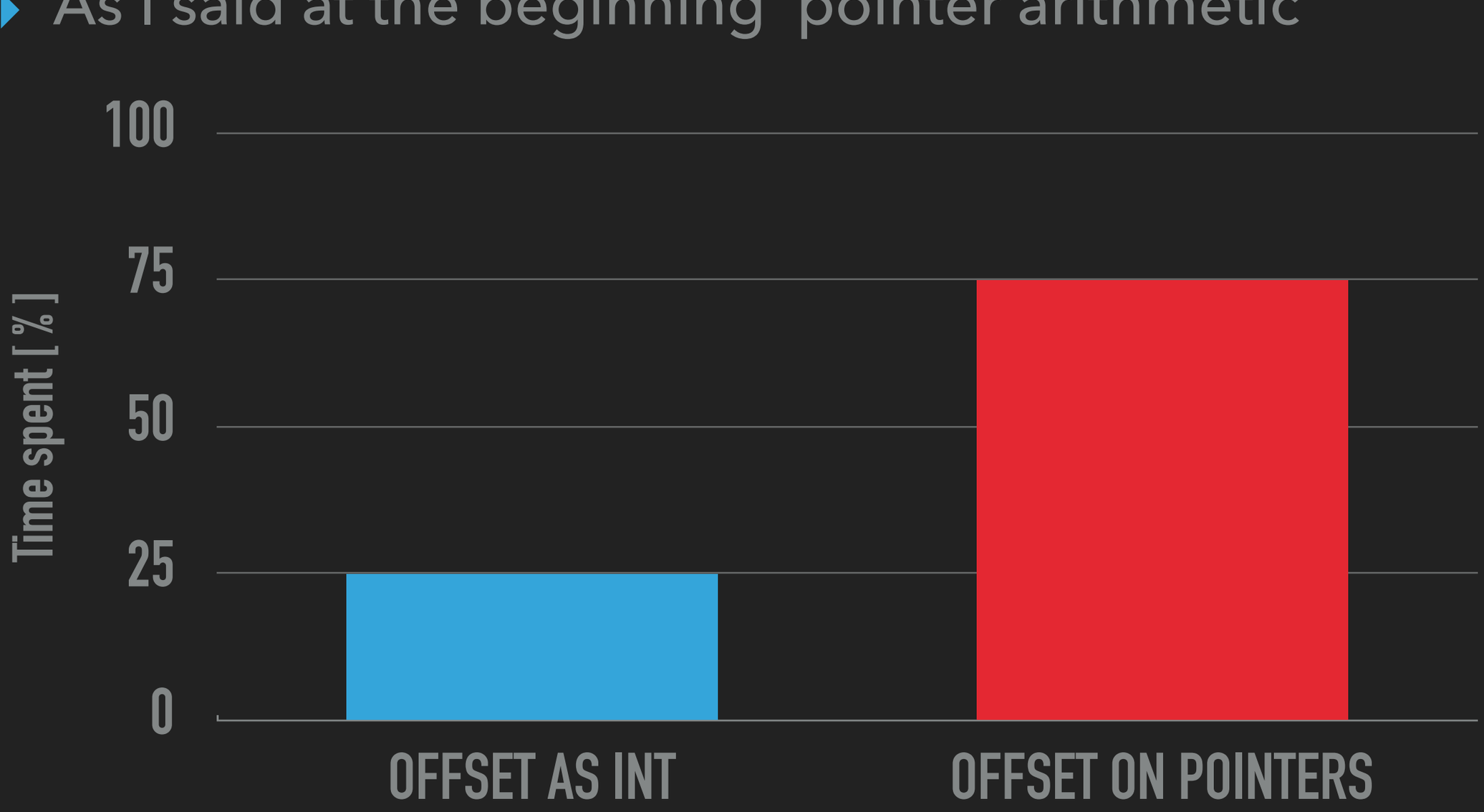
```
list.add(0);
list.add(10);
list.add(20);
list.add(30);
list.add(40);
list.add(50);
list.add(60);
list.add(70);
list.print();
std::cout << "list.exists(20) == " << list.exists(20) << std::endl;
list.remove(40);
list.remove(50);
list.add(150);
list.add(140);
list.remove(50);
list.add(160);
list.print();
shmBlock::freeShm(shmBlockSize);
```

```
Memory block of size 8192 allocated at: 0x10ced2000
List capacity: 682
freeOffset = 0
freeOffset = 1
freeOffset = 2
freeOffset = 3
freeOffset = 4
freeOffset = 5
freeOffset = 6
freeOffset = 7
Current: 70
Current: 60
Current: 50
Current: 40
Current: 30
Current: 20
Current: 10
Current: 0
list.exists(20) == 1
removing node with value 40
removing node with value 50
freeOffset = 4
freeOffset = 5
freeOffset = 8
Current: 160
Current: 140
Current: 150
Current: 70
Current: 60
Current: 30
Current: 20
Current: 10
Current: 0
Memory block has been released...
```

```
Now we will read linked list from memory
Memory block of size 8192 read at: 0x10ced2000
List capacity: 682
Current: 160
Current: 140
Current: 150
Current: 70
Current: 60
Current: 30
Current: 20
Current: 10
Current: 0
Memory block has been released...
```

HARDEST ISSUE

- As I said at the beginning pointer arithmetic



THANKS FOR LISTENING