Operating Systems Lab - Week 7: exercise - with answers

This lab is about memory allocation, structures, and linked lists. In the first exercise, you learn to build a customized *allocator*. This may help with the dynamic allocation process and avoid memory leaks. In the second section, you will use a simply linked list to store words. Linked lists that can dynamically grow allow you to drop any constraints on the length of the user input.

1 Controlled memory allocations

Build a tool that prints on `stdout` the number of current *heap allocations*. The idea is to declare (in `main`) and update an *allocation counter*, `counter`, which is incremented by 1 every time you call `malloc` and is reduced by 1 every time you call `free`. More explicitly, write

- `heapAllocator`, a function that calls `malloc`, returns the pointer returned by `malloc`, and, if `malloc` returns a non-null pointer, increment the counter by 1, and
- `heapDeAllocator`, a function that calls `free` and reduces the counter by 1.

In both cases, the counter *is not* the return value of the functions.

1.1 Test program

As a test, you can use the C code of C example 7.2, available on Moodle. The program creates a linked list of `n` nodes storing one of the first `n` integers. Dynamic memory is needed because the user chooses the number of nodes at runtime. Check with Valgrind that the program does not leak memory when it runs and answer the following questions.

- In which order the nodes are printed on the screen?

  Answer: From the last node, which stores the integer `n`, to the first, storing 1.

- Is it possible to rewrite the entire code using the `dot-notation`, i.e. to remove all arrows "`->`" from the code?

  Answer: The code without arrows reads

```
#include <stdio.h>
#include<stdlib.h>
struct node{
  int val;
  struct node *next;
};
int main() {
  int n = getchar() - '0';
  struct node *head = NULL;
  for (int i=0; i < n; i++) {
    struct node *cur = malloc(sizeof(struct node));
    (*cur).val = (i+1);
    (*cur).next = head;
    head = cur;
  }
  struct node *cur = head;
  for (int i=0; i < n; i++) {
    printf("address node %d = %p\n", n-i, (void *) cur);
    printf("value node %d = %d\n", n-i, (*cur).val);
    printf("reference node %d = %p\n", n-i, (void *) (*cur).next);
```

```c
    printf("------------------------------\n");
    if (cur) cur=(*cur).next;
  }
  for (int i=0; i < n; i++) {
    struct node *cur = head;
    head  = (*cur).next;
    free(cur);
  }
}
```

- What is the output of

  ```c
  printf("head->next->next->next->next->val=%d\n",
                           head->next->next->next->next->val);
  ```

  if you insert it before and after the `for`-loop that prints the list?

  Answer: If you set `n` = 6 you get `head->next->next->next->next->val=2`

- Remove the sanity check in the last line of the printing loops, i.e. replace

  ```c
  if (cur) cur=cur->next;
  ```

  with

  ```c
  cur=cur->next;
  ```

  What happens? Can you understand why you get an execution error and the program crashes?

  Answer: When `cur` is `NULL` there is no `cur->next`

1.2 heapAllocator

The allocator is based on the memory-allocation function `malloc`. The function can be defined as

```c
void *heapAllocator(int size, int *counter) {
  void *cur = malloc(size);
  if (cur)(*counter)++;
  return cur;
}
```

Answer the following questions:

1. Why do you need a pointer to an integer as a second argument?

   Answer: As `counter` is defined in `main`, we need the function not to discard the new values.

2. How does `main` know that the counter has been updated after the function has returned?

   Answer: Because `counter` is passed to the function through its address.

3. Would the function work for allocating a node of the integer list, as in C example 7.2, and a character array?

   Answer: Yes.

1.3 `heapDeAllocator`

The de-allocator is based on the memory de-allocation function `free`. The function can be defined as

```c
void heapDeAllocator(void *p, int *counter) {
        if (p) (*counter)--;
        free(p);
}
```

Answer the following questions:

1. What is the return value of `heapDeAllocator`?

   Answer: Similarly to `free`, the function has no return value.

2. Can you explain what happens if you remove the parenthesis and the star in `(*counter)`?

   Answer: The counter is not updated because the function increments the pointer value but the changes are discarded when the function returns.

3. What happens if you call `heapDeAllocator` with a null pointer as a first argument?

   Answer: The function does not do anything.

1.4 Testing C example 7.2 for memory leaks

Use `heapAllocator` and `heapDeAllocator` defined above to check whether the program in C example 7.2 correctly frees all heap-allocated memory. Save the code of C example 7.2in a new file called `integerCheck.c` and modify the code as suggested below.

1. At the beginning of `main`, *declare* the counter, and initialize it to 0. Be sure that you set it before you start allocating memory dynamically.

2. *Replace* each call of `malloc` with a call to `heapAllocator`. Avoid compilation errors by passing the function's second argument in the right format.

3. *Replace* each call of `free` with a call to `heapDeAllocator`.

4. Print the *value of the counter* by adding the following line below each call of `heapAllocator` and `heapDeAllocator`,

   ```c
   printf("counter=%d\n", counter);
   ```

5. In the `for` loop that prints the list, *comment out* all other calls to `printf`, except for the one printing the node value of the nodes, i.e. replace

   ```c
   struct node *cur = head;
   for (int i=0; i < n; i++) {
     printf("address node %d = %p\n", n-i, (void *) cur);
     printf("value node %d = %d\n", n-i, cur->val);
     printf("reference node %d = %p\n", n-i, (void *) cur->next);
     printf("------------------------------\n");
     if (cur) cur=cur->next;
   }
   ```

   with

   ```c
   struct node *cur = head;
   for (int i=0; i < n; i++) {
     printf("value node %d = %d\n", n-i, cur->val);
     if (cur) cur=cur->next;
   }
   ```

Check that your program compiles and runs without errors, that Valgrind produces no warning messages, and that your program has the following output (if you enter 3 in the terminal after it starts).

```
4
counter=1
counter=2
counter=3
counter=4
value node 4 = 4
value node 3 = 3
value node 2 = 2
value node 1 = 1
counter=3
counter=2
counter=1
counter=0
```

Answer:

```c
#include <stdio.h>
#include<stdlib.h>
struct node{
        int val;
        struct node *next;
};
void *heapAllocator(int size, int *counter) {
        void *cur = malloc(size);
        if (cur)(*counter)++;
        return cur;
}
void heapDeAllocator(void *p, int *counter) {
        if (p) (*counter)--;
        free(p);
}
int main() {
        int counter = 0;
        int n = getchar() - '0';
        struct node *head = NULL;
        for (int i=0; i < n; i++) {
                struct node *cur = heapAllocator(sizeof(struct node), &counter);
                printf("counter=%d\n", counter);
                cur->val = (i+1) ;
                cur->next = head;
                head = cur;
        }
        struct node *cur = head;
        for (int i=0; i < n; i++) {
                printf("value node %d = %d\n", n-i, cur->val);
                if (cur) cur=cur->next;
        }
        for (int i=0; i < n; i++) {
                struct node *cur = head;
                head  = cur->next;
                heapDeAllocator(cur, &counter);
                printf("counter=%d\n", counter);
        }
}
```

2 A simply linked list of strings

In this section, you will write a program that creates a linked list to store a series of words entered by the user. The idea is

- to isolate single words of the input iteratively, by stopping reading characters when you reach `' '`,
- to allocate a new node of the list for each new word,
- to store the words and their length in the nodes, and
- to stop parsing the input when you reach a new line character.

You will need a series of subroutines given in Section 2.6. The structure of the main program is also given and you only need to complete a few statements. Save your code into a file called `linkedWords.c` and do not forget to check your code with Valgrind to see if it runs correctly.

2.1 Define a node

Avoid overflow problems in a structure definition, by fixing the maximum number of characters stored in a node. Let `MAX` be a macro defined at the beginning of the program as

```
#define MAX 10
```

and each node be an instance of

```
struct node {
    int length;
    char word[MAX];
    struct node *next;
};
```

where the `length` will store the string length, `word` the input word, and `next` the pointer to the next node.

2.2 Parsing the input

The input is processed through the function

```
{\tt int getWord(char *buf, int *end, int maxLength)}
```

that

- copies a single word from `stdin` into `buf`,
- sets `end` to 1 at the end of the `stdin` input, and
- returns the number of processed characters.

The number of processed characters is used to check if the *entire* word can be stored in the buffer and if a *valid* word has been found.

2.3 Creating a node

To store the string into a node, you need to

- allocate a new node using `malloc`,
- copy the string into the node's character array by calling,

  ```
  int copyString(char *in, char *out)
  ```

  which copies the content of `in` into `out` and returns the length of the string,

- stores the length of the string, obtained from

  ```
  int stringLength(char *s)
  ```

  that returns the number of characters in `s`, in the node's integer,
- link the current node to the *head* of the list, and
- move the head to the current node.

The memory allocation and the linking strategy are similar to the case of a list of integers.

2.4 Printing and freeing the list

To print the string iteratively, you can use

```
int printList(struct node *head, int n)
```

that prints the content of a list of `n` nodes, starting from the node pointed by `head`. To free the list, you can use

```
int freeList(struct node *head)
```

that frees the memory allocated for each node, starting from the node pointed by `head`.

2.5 Write `main`

As `main`, you can use the following template

```
int main() {
  char buf[MAX];
  struct node *head = ...
  struct node *cur = ...
  int end = 0;
  int count = 0;
  printf("enter words:\n");
  while (end == ...) {
    int j = getWord(...);
    if (j > 0) {
      buf[j]= ... ;
      cur = malloc(...);
      cur->next = ...;
      copyString(...);
      cur->length = stringLength(...);
      head = ...
      count = ...;
    }
  }
  int iPrint = printList(...);
  int iFree = freeList(...);
  printf("(count, iPrint, iFree)=(%d, %d, %d)\n", count, iPrint, iFree);
  return 0;
}
```

2.6 Input-parsing and string-handling

```
int stringLength(char *s) {
  int i = 0;
  while (s[i] != '\0') i++;
  return i;
}
```

```
int copyString(char *in, char *out) {
  int i = 0;
  while (in[i] != '\0') {
    out[i] = in[i];
    i++;
  }
  out[i]='\0';
  return i;
}
```

```
int getWord(char *buf, int *end, int maxLength) {
  int j = 0;
  char c = '\0';
  while (((c = getchar()) != ' ') && (c != '\n') && (j < maxLength))
    buf[j++]=c;
  if (j == maxLength) buf[j++] = c;
  if (c == '\n') *end = 1;
  return j;
}
```

```
int printList(struct node *head, int n) {
  struct node *iter = head;
  int i = 0;
  while (iter) {
    printf("%d-th node: %s (%d)\n", n - i, iter->word, iter->length);
    iter = iter->next;
    i++;
    }
    return i;
}
```

```
int freeList(struct node *head) {
  struct node *iter = head;
  int i = 0;
  while (head != NULL) {
    iter = (*head).next;
    free(head);
    head = iter;
    i++;
  }
  return i;
}
```

2.7 Example

A run of the program with user input,

```
one Two three FourFiveSix7Eight nine      Ten
```

should produce the following output.

```
./a.out
enter words:
one Two three FourFiveSix7Eight nine      Ten
7-th node: Ten (3)
6-th node: nine (4)
5-th node: x7Eight (7)
4-th node: FourFiveSi (10)
3-th node: three (5)
```

```
2-th node: Two (3)
1-th node: one (3)
(count, iPrint, iFree)=(7, 7, 7)
```

If you execute your program with Valgrind (and the same **stdin** input as above), you should not see any error or leaking message and print something similar to

```
valgrind ./a.out
==2239488== Memcheck, a memory error detector
==2239488== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2239488== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2239488== Command: ./a.out
==2239488==
enter words:
one Two three FourFiveSix7Eight nine      Ten
7-th node: Ten (3)
6-th node: nine (4)
5-th node: x7Eight (7)
4-th node: FourFiveSi (10)
3-th node: three (5)
2-th node: Two (3)
1-th node: one (3)
(count, iPrint, iFree)=(7, 7, 7)
==2239488==
==2239488== HEAP SUMMARY:
==2239488==     in use at exit: 0 bytes in 0 blocks
==2239488==   total heap usage: 9 allocs, 9 frees, 2,216 bytes allocated
==2239488==
==2239488== All heap blocks were freed -- no leaks are possible
==2239488==
==2239488== For lists of detected and suppressed errors, rerun with: -s
==2239488== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Answer:

```c
int main() {
  char buf[MAX];
  struct node *head = NULL;
  struct node *cur = NULL;
  int end = 0;
  int count = 0;
  printf("enter words:\n");
  while (end == 0) {
    int j = getWord(buf, &end, MAX - 1);
    if (j > 0) {
      buf[j]='\0';
      cur = malloc(sizeof(struct node));
      cur->next = head;
      copyString(buf, cur->word);
            cur->length = stringLength(buf);
      head = cur;
      count++;
    }
  }
  int iPrint = printList(head, count);
  int iFree = freeList(head);
  printf("(count, iPrint, iFree)=(%d, %d, %d)\n", count, iPrint, iFree);
  return 0;
}
```