CS2850 Operating System Lab

Week 6: Dynamic memory

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

outline

The heap and the stack

Static memory allocation

Dynamic memory allocation

Processes

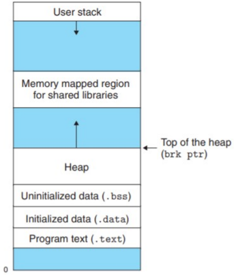
A process is a *running program*.

Each process has a private address space, i.e. a set of memory locations *labeled* from 0 to some maximum.

The address space contains the program

- *executable file*,
- *data*,
- *stack*, and
- *heap*.

Virtual address space



The process *virtual* address space is a list of memory locations.

The program reads from and writes to both the heap and the stack.

The stack

The stack is the region *on the top* of the virtual address space used to implement *function calls*.

It *expands and contracts* when the program runs, e.g.

- it grows each time a function is called, e.g. to allocate *local variables*, and
- it contracts when a function returns as all local variables are *discarded*.

The heap

The heap is a *collection* of various-sized blocks, each block being a slot of virtual memory that is either allocated or free.

The code and data areas are *just below* the run-time heap.

The heap *expands and contracts* dynamically at run time as dynamic memory is allocated.

The kernel maintains a variable that points to the *top of the heap*.

Example

Run this program to check that *static* and *dynamic* objects are allocated in different address space regions.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        char a = 'a';
        char b[10];
        char *aDyn=malloc(sizeof(char));
        *aDyn = 'A';
        char *bDyn=malloc(10 * sizeof(char));
        for (int i=0; i<10; i++) {
                *(b + i) = 'b';
                *(bDyn + i) = 'B';
        }
        printf("a=%c and &a=%p\n", a, (void *) &a);
        printf("b[0]=%c and &b[0]=%p\n", *b, (void *) b);
        printf("*aDyn=%c and &p[0]=%p\n", *aDyn, (void *) aDyn);
        printf("bDyn[0]=%c and &bDyn[0]=%p\n", *bDyn, (void *) bDyn);
        free(aDyn);
        free(bDyn);
}
```

Static memory allocation

In C, the amount of memory used to allocate variables in the stack is determined at compile time.

Stack variables are declared by writing

```
char a;
int b[10];
```

The size of a stack variable can not be changed *at runtime*. In other words, the memory allocation is static.

Dynamic memory allocation

The standard library provides functions for allocating memory in the run-time heap.

Heap memory blocks are allocated dynamically and it is possible to change the size of the corresponding variables.

To change the size of a heap-allocated variable, you need to call a library function that *finds* a new suitable slot and *frees* the old one.

Memory allocator

The library function for allocating memory in the heap is

```
void *malloc(size_t size)
```

`size_t` is the size of the block to be allocated.

For example, you can allocate a *memory region* for hosting an array of 10 integers with

```
int *a;
a = malloc(10 * sizeof(int))
```

`malloc`

`malloc` is defined `stdlib.h` and

- returns a `void` pointer [1], if the allocation is successful, and `NULL` otherwise, and
- creates a new un-typed and uninitialised block of memory of size `size`.

After allocating the right amount of memory you can cast the `void` pointer to any required type.

[1] A pointer to unspecified data

How not to write your program

"ISO C99 introduced the capability to have array dimensions be expressions that are computed as the array is being allocated, and recent versions of `gcc` *support most of the conventions for variable-sized arrays in ISO C99.*[2]

In this course, you should ignore these extensions and always use `malloc` to allocate memory dynamically.

[2] from Section 3.8.5 of Computer Systems: A Programmer's Perspective C

`realloc` and `free`

Other memory-allocation functions defined in `stdlib.h` are

- `void *realloc(void *p, size_t size)`, which changes the size of the object pointed to by `p` to a new given size, `size`, without changing its content (up to `size`), and
- `void free(void *p)`, which de-allocates the *heap space* pointed to by `p`.

The argument of `free` must be a pointer to *previously allocated dynamic memory*. For example, the following statements cause an *execution error*.

```
int i = 1;
free(i);
```

Free the memory

It is good practice to *free any dynamically allocated memory that is no longer needed* because

- memory usage keeps growing with every new allocation and
- it may help avoid memory leaks, i.e. allocated memory slots that are no longer referenced by a pointer.

Suggestion: Always check if your program leaks memory by running

```
valgrind ./a.out
```

Example

This program produces a memory leak of 40 bytes because the corresponding pointer is *moved elsewhere*.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int i = 1;
  int *a =malloc(10 * sizeof(int));
  for (int j=0; j<10; j++)
    *(a+j)=j;
  a = &i;
  printf("a[9]=%d\n", *(a+9));
  //free(a);
}
```

Example

Why does running the program (without the final call to `free`) produce a *strange* output?

```
./a.out
a[9]=-1701673536
```

If you *un-comment* the last line makes the program crash and you get

```
./a.out
a[9]=1133786560
free(): invalid pointer
Aborted (core dumped)
```

valgrind

Running the program with `valgrind` may help you spot the issue as you get

```
valgrind ./a.out
==1570619== Memcheck, a memory error detector
...
==1570619== Command: ./a.out
a[9]=0
==1570619== HEAP SUMMARY:
==1570619==     in use at exit: 40 bytes in 1 blocks
==1570619==   total heap usage: 2 allocs, 1 frees, 1,064 bytes allocated
==1570619== LEAK SUMMARY:
==1570619==    definitely lost: 40 bytes in 1 blocks
==1570619==    indirectly lost: 0 bytes in 0 blocks
==1570619==      possibly lost: 0 bytes in 0 blocks
==1570619==    still reachable: 0 bytes in 0 blocks
==1570619==         suppressed: 0 bytes in 0 blocks
...
==1570619== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

references

Read more about the `malloc` family on

- this page of the online C manual,
- Section 8.7 of The C Programming Langauge, or
- Section 9.9 of Computer Systems: A Programmer's Perspective C.