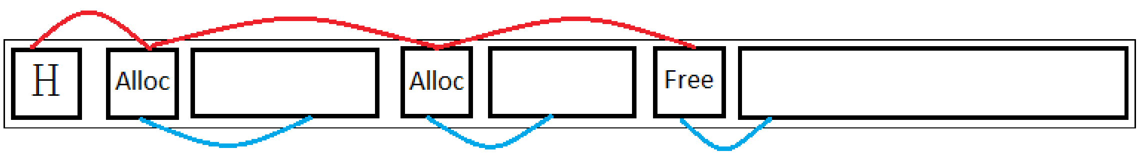
Serialize your Memory

Memory serialization program for C language

Introduction

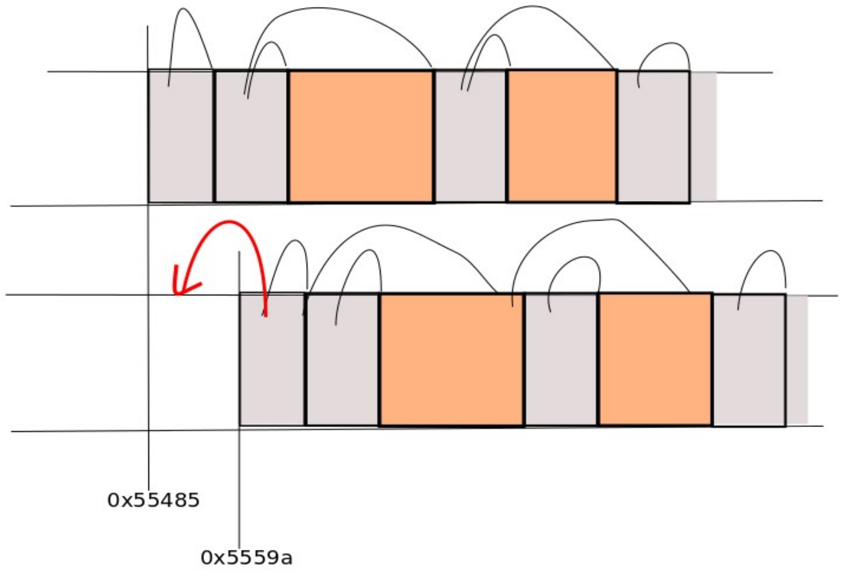
This is the documentation of C memory serialization method by using extended malloc and extended mmap. You already know the famous malloc and free memory allocation and deallocation functions in C language. And if you were aware of memory serialization (storing data in the RAM in a file) or memory hibernation, you should familiar with mmap and munmap. This implementation is a combination of these two concepts. In here mmap and munmap functions are used straight away. But the concept behind malloc and free are implemented in a different way to overcome physical memory conflict problems.

The very fundamental concept which used in here is the process of mmap and munmap functions (system calls). mmap function can write an array of data in to a file and munmap function can load that data again in to memory. It is the only serialization or hibernation facility which C libraries have. It can not be done write a tree or linked list in to a file and reuse it. That problem targets here. It is better to find the process of malloc and free.



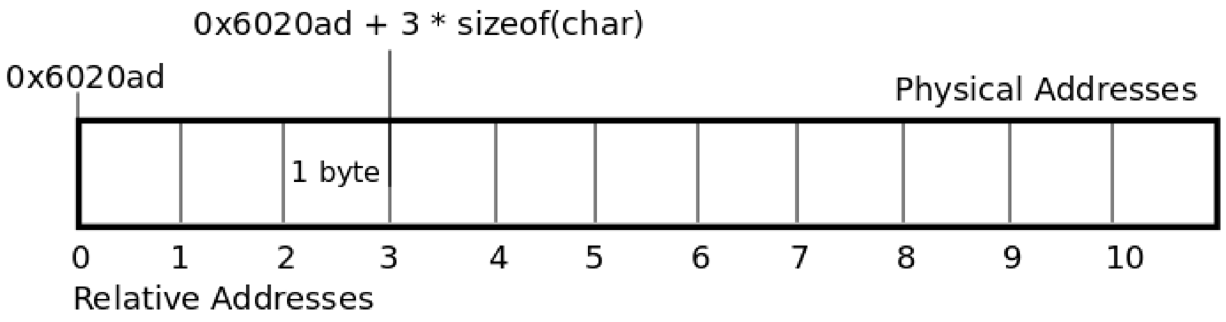
This diagram shows the implementation of malloc by using first-fit algorithm. Each node (in here, meta data) points to a reference data space and the next node block. What are these pointers. Not like other languages, in C these pointers are physical memory location which saved in a pointer variable. They are just memory addresses, just bunch of values. That's the ever lasting beauty and almost the ugly in C programming. I have an array of memory and by using my own implementation of malloc and free (in here, it's malloc_e and free_e), I also have a flat array of memory of any kind of data structure. It could be a Tree, a Linked List or a Graph. I don't care, since all those things are under my control, inside my array of memory. Since memory is allocated from the stack, at the end of the program we don't have to worry about freeing them.

So here we go. Now we have a solution. We can implement serialization program. I did it. It works very well. But suddenly when I compile it and run it on Arch, it didn't work. OMG! What's now. That is another crazy thing in C language. The undefined situations. I've used memory addresses and store those addresses in a file. But when I retrieve it, the starting address of my array is shifted. In this scenario, none of the pointer will work properly. Because your stored memory addresses are pointed to somewhere else. It refers the same memory location, when it used at the time of serialization. But now memory is changed. And my program can not access those memory locations any more.



Can you see the problem? It is in the red coloured reference line. With the new memory locations, it refers a wrong location. Problem is not in the pointer. But it is in the value of pointer. We hope that, our memory will be arranged like black reference lines in the second diagram. But it's not. All the pointers are pointed to a wrong locations. One solution is shift all the pointers by some value (i.e. 0x5559a – 0x55485). It works for the read only memory serialization. But once we edit the unmapped memory file, there can be another set of memory addresses which belongs to a another base address. Next time you have to manage all these two sets of memory locations. What is the scenario after 10 times of serializations. You can not manage it. So this is a solution for the read only memory serializations. But not for the read and write.

We have to think like a OS. How does OS manage it's memory. Especially the virtual memory. OS does memory management for my program and my program also do memory management for it's clients. Two levels of virtual addresses. Why I told that. OS tells to it's programs that, "all the memory is yours and it has a liner address space". But really it's not. It's not linear and all the memory is not mine. Same here. The condition for a memory serializations is, **don't use pointers straight away**. That's the way it overcomes this problem. Serialization program uses virtual addresses (actually relative address) as it's pointers. And these addresses are **unsigned int** numbers.



API

`void* malloc_e (size_t size)`	Allocate memory block from predefined memory space and return a void pointer of that memory location
`void free_e (void* ptr)`	Change the state of memory block into the free state and do de-fragmentation
`void mmap_e (char* filename)`	Write whole memory space into a file
`void* munmap_e (char* filename)`	Load data from the file, set data structure which used to memory management and return a void pointer of first allocated memory chunk
`void putString (char* p, const char* text)`	Assign data into a char pointer (assign text data), if client's data structure uses char*
`void* PA (int virAddr)`	Give the physical address (pointer) of any given relative (virtual) memory address
`unsigned int VA (void* phyAddr)`	Give the relative (virtual) address of any given void pointer
`void printList ()`	Print the details of memory management data structure
`void defragmentFreeNodes ()`	De-fragmentate predefined memory space (attach with inner process, no need of doing this manually)

Inner defines of the program. Not visible to the client.

`#define NUMINTS (50000)`	Size of the predefined memory space
`#define P (virAddr)`	Give the physical address (pointer) of any given relative (virtual) memory address
`#define V (phyAddr)`	Give the relative (virtual) address of any given void pointer

Requirements for serialization

- **There can not be pointers** in the data structure. That means, you should not store pointers in the predefined memory space which provides by malloc_e ()
- Since it is not allowed to store pointers, you have to use **unsigned int** to store virtual addresses. And virtual - physical and physical - virtual conversion can be done by using PA() and VA() inbuilt functions.
- It is recommended to **use your own defines** to manage your data structure and char*

  i.e.

```
#define SP(p) (char*)(PA(p)) //SP for string pointers
#define BP(p) ((Box*)(PA(p))) //BP for Box pointers; Your data structure
```

Example program

```
#include "serialize.h"

#define SP(p) (char*)(PA(p))
#define BP(p) ((Box*)(PA(p)))

typedef struct box
{
      unsigned int name;
      int age;
      unsigned int next;
}Box;

int main()
{
      Box* head = malloc_e(sizeof(Box));

      head->name = VA(malloc_e(sizeof(char)*10));
      putString(SP(head->name), "amith\0");
      head->age = 23;
      head->next = VA(malloc_e(sizeof(Box)));

      printf("%s:%d\n", SP(head->name), head->age);
      printList();

      mmap_e("memFile.bin");

      head = (Box*)munmap_e("memFile.bin");

      BP(head->next)->name = VA(malloc_e(sizeof(char)*10));
      putString(SP(BP(head->next)->name), "Rechel\0");
      BP(head->next)->age = 25;

      printf("%s:%d\n", SP(BP(head->next)->name), BP(head->next)->age);

      printList();

      return 0;
}
```

Distribution terms

References

- mmap and munmap code examples which used to build mmap_e and munmap_e

  http://www.linuxquestions.org/questions/programming-9/mmap-tutorial-c-c-511265/

  Hko
  Senior Member