CS342 Operating Systems - Spring 2021
Project #3 – Memory Management

Assigned: Apr 3, 2021, Saturday.
Due date: Apr 18, 2021, Sunday, 23:59. Document version: 1.0

- *The project can be done in groups of 2 or individually.*

Assignment

In this project you will implement a memory management library called **sbmem.a** (buddy memory allocation from shared memory). Processes will use your library to allocate memory space dynamically, instead of using the well-known malloc function. A shared memory object, i.e., a shared memory segment, needs to be created first. Memory space allocations will be made from that segment to the requesting processes. Your library will implement the necessary initialization, allocation and deallocation routines. Your library will keep track of the allocated and free spaces in the memory segment. For that it will use the Buddy memory allocation algoritm [1, 2]. Study buddy algorithm first. It is and old and elegant algorithm. It is easy to grasp.

Your library will implement the following functions. The first two functions will be used by programs that will create or destroy the shared segment to be used by other processes.

- int sbmem_init (int segsize). This function will create and initialize a shared memory segment of the given size. The given size is in bytes and must be a power of 2. Memory will be allocated from that segment to the requesting processes later. If operation is successful, the function will return 0, otherwise, it will return -1. You will use POSIX shm_open() and ftruncate() functions in the implementation of this function. If there is already a shared segment, the function will destroy the segment first, before creating a new one with the same name. After initialization, the shared segment will be ready to use. That means processes can make memory allocation requests, and memory can be allocated from this segment, if available.

- sbmem_remove (). This function will remove the shared memory segment from the system. You will use shm_unlink () function to implement this. The function will do all the necessary cleanup. For example, the created semaphore(s) will be removed as well.

A process that would like to use the library for memory allocations will use the following functions that you will implement in your library (**sbmemlib.c**).

- int sbmem_open(). This function will indicate to the library that the process would like to use the library. In this way, the library can keep track of the processes that are interested in using the library. The library will map the shared segment to the virtual address space of the process using the mmap function. If

there are two many processes using the library at the moment, sem_open will return -1. Otherwise, if the process can use the library, sem_open will return 0.

- void *sbmem_alloc (int reqsize). This function will allocate memory of size n, which is the next power of 2 greater than or equal to reqsize. The reqsize is in bytes. For example, if reqsize is 200 (bytes), then the library will allocate memory of size 256, because 256 is the next power of 2 greater than or equal to 200. If allocation succeeds, the function will return a pointer to the allocated space. It is up to the program what to store in the allocated space. NULL will be returned if memory could not be allocated. This can happen, for example, when there is not enough memory.

- void sbmem_free (void *ptr). This function will deallocate the memory space allocated earlier and pointed by the pointer ptr. The deallocated memory space will be part of the free memory in the segment.

- int sbmem_close (). When a process has finished using the library it will call sbmem_close(). In this way your library will know that this process will not use the library for memory allocation, hence shared segment can be unmapped from the virtual address space of the process. If the process would like to use the library again, it should call sbmem_open again. Normally you need to deallocate all the space allocated earlier by the process when the process calls sbmem_close() (if the process did not free already). But to reduce bookkeeping, deallocating all space allocated earlier by the process when the process call sbmem_close() is OPTIONAL for the project . You don't have to do it. In our tests, we will free each memory chunk that we allocated earlier before calling sbmem_close().

You will use semaphore(s) in your library to protect your shared structures. Processes may call allocation and free functions concurrently. This should not cause race conditions. You will ensure that by using semaphores.

A program, for example app.c, that wants to use your library for memory allocations may use the related functions as below. The program should include the library header file **smem.h**. This header file is the interface of the library.

```
include <unistd.h>
#include <stdlib.h>

#include "sbmem.h"
#define ASIZE 256

int main()
{
    int i, ret;
    char *p;

    ret = sbmem_open();
    if (ret == -1)
         exit (1);

    p = sbmem_alloc (ASIZE); // allocate space to for characters
    for (i = 0; i < ASIZE; ++i)
         p[i] = 'a'; // init all chars to ‘a’
    sbmem_free (p);
```

```
    sbmem_close();
    return (0);
}
```

You will also implement two simple programs that will be called as create_memory.c and destroy_memory.c. The create_memory program will basically call the sbmem_init function of your library to create and initialize a shared memory segment. The destroy_mem program will basically call your sbmem_remove function to remove the shared segment from system.

We will test your library with various programs that we will implement using your library.

Experiments and Report (20 pts)

Do a lot of experiments to measure fragmentation amount. Plot your results. Draw conclusions. Write a report to include all the experiment descriptions and results.

Submission

You will submit the source files (C and header files) of your programs and also a Makefile and a README file. Put all these files into a directory named with your Student ID, and tar and gzip the directory. For example a student with ID 21404312 will create a directory named "21404312" and will put the files there. Then he/she will tar the directory (package the directory) as follows:

tar cvf 21404312.tar 21404312

Then he/she will gzip the tar file as follows:

gzip 21404312.tar

In this way he/she will obtain a file called 21404312.tar.gz. Then he/she will upload this file in Moodle. For a group, one submission with a single ID (the ID of either student) is enough.

References

[1] Buddy Memory Allocation.
https://en.wikipedia.org/wiki/Buddy_memory_allocation

[2] The Art of Programming. Donald Knuth.

[3] There are a lot resources in Internet explaining POSIX shared memory, and POSIX semaphores. If you wish, you can start with reading the related Linux manual pages. For that, type: "man shm_overview" and "man sem_overview".

Tips and Clarifications

- Start early.
- You will use POSIX shared memory API and POSIX semaphores API.
- At most 10 processes will request memory simultaneously.
- The minimum shared segment size is 32 KB. The maximum size is 256 KB.
- The minimum request size is 128 bytes. The maximum request size is 4096.

- A git repository is created including sample files to start with. You can start with them.

  https://github.com/korpeoglu/cs342spring2021-p3
- Note that for each different process, the shared segment might have been mapped to a different location in its virtual address space. Hence when a pointer is given to to the library by a process, or when the library gives a pointer to a process, you should convert the pointer to a standard value or vice versa, i.e., normalize.
- In other words, each process will have its own start address (pointer) to the shared segment. In the shared segment, if you are storing an address, that can be (should be) a relative address, assuming that the first byte of the shared segment has relative address (offset) 0. Hence, in the segment you need to store offset values (not virtual addresses of the processes).
- You can use the IDs of the processes to uniquely identify them inside your library. You can learn the id of a process by using the getpid() system call. For example, when a process calls the sbmem_alloc function, your library, can understand which process has called the function by using the getpid system call. You may keep a table in the shared memory segment to store some information about the processes that are using the library at the moment (i.e., that called the sbmem_open) function.
- Shared segment will have persistence across processes, but the library not.
- Note that allocation information will be kept as part of the shared memory chunks as well. You will not use a separate structure or memory to keep the allocation state of the shared segment.