C Programming

How to use mmap function in C language?

6 months ago • by Bamdeb Ghosh

The ***mmap()*** function is used for mapping between a process address space and either files or devices. When a file is mapped to a process address space, the file can be accessed like an array in the program. This is one of the most efficient ways to access data in the file and provides a seamless coding interface that is natural for a data structure that can be assessed without he abstraction of reading and writing from files. In this article, we are going to discuss how to use the ***mmap()*** function in Linux. So, let's get started.

Header File:

```
#include <sys/mman.h>
```

Syntax:

```
void * mmap (void *address, size_t length, int protect, int flags, int filedes,
off_t offset)
```

Arguments:

The function takes 6 arguments:

1. address:

This argument gives a preferred starting address for the mapping. If another mapping does not exist there, then the kernel will pick a nearby page boundary and create the mapping; otherwise, the kernel picks a new address. If this argument is NULL, then the kernel can place the mapping anywhere it sees fit.

2. length:

This is the number of bytes which to be mapped.

3. protect:

This argument is used to control what kind of access is permitted. This argument may be logical 'OR' of the following flags ***PROT_READ | PROT_WRITE | PROT_EXEC | PROT_NONE.*** The access types of read, write and execute are the permissions on the content.

4. flags:

This argument is used to control the nature of the map. Following are some common values of the flags:

- ***MAP_SHARED:*** This flag is used to share the mapping with all other processes, which are mapped to this object. Changes made to the mapping region will be written back to the file.
- ***MAP_PRIVATE:*** When this flag is used, the mapping will not be seen by any other processes, and the changes made will not be written to the file.
- ***MAP_ANONYMOUS / MAP_ANON:*** This flag is used to create an anonymous mapping. Anonymous mapping means the mapping is not connected to any files. This mapping is used as the basic primitive to extend the heap.
- ***MAP_FIXED:*** When this flag is used, the system has to be forced to use the exact mapping address specified in the *address* If this is not possible, then the mapping will be failed.

5. filedes:

This is the file descriptor which has to be mapped.

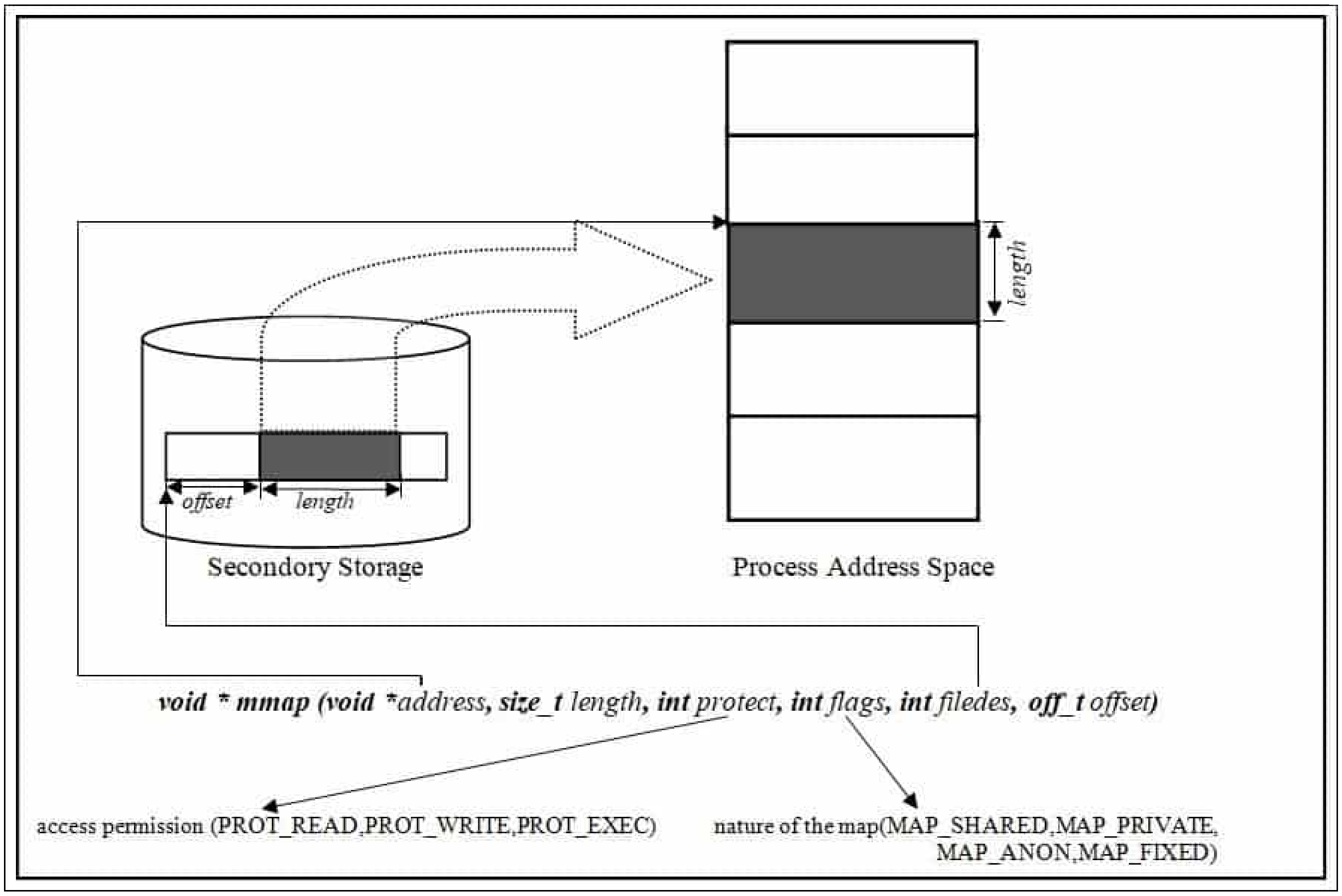
6. offset:

This is offset from where the file mapping started. In simple terms, the mapping connects to ***(offset)*** to ***(offset+length-1)*** bytes for the file open on ***filedes*** descriptor.

Return values:

On success, the ***mmap()*** returns 0; for failure, the function returns MAP_FAILED.

Pictorially, we can represent the map function as follows:


offset
length
Secondory Storage
Process Address Space
length
void * mmap (void *address, size_t length, int protect, int flags, int filedes, off_t offset)
access permission (PROT_READ,PROT_WRITE,PROT_EXEC)
nature of the map(MAP_SHARED,MAP_PRIVATE, MAP_ANON,MAP_FIXED)

For unmap the mapped region ***munmap()*** function is used :

Syntax:

int munmap(***void*** **address**, ***size_t*** *length*);

Return values:

On success, the ***munmap()*** returns 0; for failure, the function returns -1.

Examples:

Now we will see an example program for each of the following using mmap() system call:

- Memory allocation (Example1.c)
- Reading file (Example2.c)
- Writing file (Example3.c)
- Interprocess communication (Example4.c)

Example1.c

```
#include <stdio.h>
#include <sys/mman.h>

int main(){

    int N=5; // Number of elements for the array
    int *ptr = mmap ( NULL, N*sizeof(int),
            PROT_READ | PROT_WRITE,
            MAP_PRIVATE | MAP_ANONYMOUS,
            0, 0 );
    if(ptr == MAP_FAILED){
        printf("Mapping Failed\n");
        return 1;
    }

    // Fill the elements of the array
    for(int i=0; i ");
    for(int i=0; i<N; i++){
        printf("[%d] ",ptr[i]);
    }

    printf("\n");
    int err = munmap(ptr, 10*sizeof(int));

    if(err != 0){
        printf("UnMapping Failed\n");
        return 1;
    }
    return 0;
}
```

In Example1.c we allocate memory using mmap. Here we used PROT_READ | PROT_WRITE protection for reading and writing to the mapped region. We used the MAP_PRIVATE | MAP_ANONYMOUS flag. MAP_PRIVATE is used because the mapping region is not shared with other processes, and MAP_ANONYMOUS is used because here, we have not mapped any file. For the same reason, the *file descriptor* and the *offset* value is set to 0.

Example2.c

```
#include <stdio.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[]){

    if(argc < 2){
        printf("File path not mentioned\n");
        exit(0);
    }

    const char *filepath = argv[1];
    int fd = open(filepath, O_RDONLY);
    if(fd < 0){
        printf("\n\"%s \" could not open\n",
                filepath);
        exit(1);
    }

    struct stat statbuf;
    int err = fstat(fd, &statbuf);
    if(err < 0){
        printf("\n\"%s \" could not open\n",
                        filepath);
        exit(2);
    }

    char *ptr = mmap(NULL,statbuf.st_size,
            PROT_READ|PROT_WRITE,MAP_SHARED,
            fd,0);
    if(ptr == MAP_FAILED){
        printf("Mapping Failed\n");
        return 1;
    }
    close(fd);

    ssize_t n = write(1,ptr,statbuf.st_size);
    if(n != statbuf.st_size){
        printf("Write failed");
    }



    err = munmap(ptr, statbuf.st_size);

    if(err != 0){
        printf("UnMapping Failed\n");
        return 1;
    }
    return 0;
}
```

In Example2.c we have mapped the file “file1.txt”. First, we have created the file, then mapped the file with the process. We open the file in O_RDONLY mode because here, we only want to read the file.

Example3.c

```
#include <stdio.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[]){

    if(argc < 2){
        printf("File path not mentioned\n");
        exit(0);
    }

    const char *filepath = argv[1];
    int fd = open(filepath, O_RDWR);
    if(fd < 0){
        printf("\n\"%s \" could not open\n",
               filepath);
        exit(1);
    }

    struct stat statbuf;
    int err = fstat(fd, &statbuf);
    if(err < 0){
        printf("\n\"%s \" could not open\n",
                       filepath);
        exit(2);
    }

    char *ptr = mmap(NULL,statbuf.st_size,
            PROT_READ|PROT_WRITE,
                        MAP_SHARED,
            fd,0);
    if(ptr == MAP_FAILED){
        printf("Mapping Failed\n");
        return 1;
    }
    close(fd);

    ssize_t n = write(1,ptr,statbuf.st_size);
    if(n != statbuf.st_size){
        printf("Write failed\n");
    }


    // Reverse the file contents
    for(size_t i=0; i \n");
    n = write(1,ptr,statbuf.st_size);
    if(n != statbuf.st_size){
        printf("Write failed\n");
    }

    err = munmap(ptr, statbuf.st_size);

    if(err != 0){
        printf("UnMapping Failed\n");
        return 1;
    }
    return 0;
}
```

In Example3.c we have read and then write to the file.

Example4.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/mman.h>

int main(){

    int N=5; // Number of elements for the array

    int *ptr = mmap(NULL,N*sizeof(int),
     PROT_READ | PROT_WRITE,
     MAP_SHARED | MAP_ANONYMOUS,
     0,0);

    if(ptr == MAP_FAILED){
     printf("Mapping Failed\n");
     return 1;
    }

    for(int i=0; i < N; i++){
     ptr[i] = i + 1;
    }

    printf("Initial values of the array elements :\n");
    for (int i = 0; i < N; i++ ){
     printf(" %d", ptr[i] );
    }
    printf("\n");

    pid_t child_pid = fork();

    if ( child_pid == 0 ){
     //child
     for (int i = 0; i < N; i++){
         ptr[i] = ptr[i] * 10;
     }
    }
    else{
     //parent
     waitpid ( child_pid, NULL, 0);
     printf("\nParent:\n");

     printf("Updated values of the array elements :\n");
     for (int i = 0; i < N; i++ ){
         printf(" %d", ptr[i] );
     }
     printf("\n");
    }

    int err = munmap(ptr, N*sizeof(int));

    if(err != 0){
     printf("UnMapping Failed\n");
     return 1;
    }
    return 0;
}
```

In Example4.c first the array is initialized with some values, then the child process updates the values. The parent process reads the values updated by the child because the mapped memory is shared by both processes.

Conclusion:

The mmap() is a powerful system call. This function should not be used when there are portability issues because this function is only supported by the Linux environment**.**

ABOUT THE AUTHOR

Bamdeb Ghosh

Bamdeb Ghosh is having hands-on experience in Wireless networking domain.He's an expert in Wireshark capture analysis on Wireless or Wired Networking along with knowledge of Android, Bluetooth, Linux commands and python. Follow his site: wifisharks.com

View all posts

RELATED LINUX HINT POSTS

- POSIX Read Function in C Programing
- POSIX Socket with C Programming
- POSIX Semaphores with C Programming
- Posix Mutex with C Programming
- 2D Array
- How Memset Function is Used
- Static in C Programming