Memory mapping

Memory mapping

Memory mapping Multiple processes can access (read/write) to a common memory which is backed by a regular file on file-system.

Advantages:

- Byte level (read/write) access without the overhead of (read(2)/write(2)) system calls.
- Multiple buffering problem is prevented.

Rules:

- Mapped region size should be OS page size multiple.
- Mapped offset should be OS page aligned.

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping.

mmap

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

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Parameter	Direction	Description
addr	in	NULL to let kernel decide the starting address or non NULL to request kernel use this as starting address for the new mapping.
length	in	length of the mapping.
prot	in	memory protection of the mapping.
flags	in	The flags argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file
fd	in	the argument to be passed to *start_routine()*
offset	in	offset in the file which needs to be mapped. length bytes will be mapped

Description:

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping.

If addr is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping. If addr is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary

The contents of a file mapping (as opposed to an anonymous mapping; see MAP_ANONYMOUS below), are initialized using length bytes starting at offset offset in the file (or other object) referred to by the file descriptor fd. offset must be a multiple of the page size as returned by sysconf(_SC_PAGE_SIZE).

The prot argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either PROT_NONE or the bitwise OR of one or more of the following flags:

PROT_EXEC	Pages may be executed.
PROT_READ	Pages may be read.
PROT_WRITE	Pages may be written.
PROT_NONE	Pages may not be accessed.

This behavior is determined by including exactly one of the following values in flags:

MAP_SHARED Share this mapping. Updates to the mapping are visible to other processes mapping the same region, and (in the case of file-backed mappings) are carried through to the underlying file. (To precisely control when updates are carried through to the underlying file requires the use of msync(2).)

MAP_PRIVATE Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the mmap() call are visible in the mapped region.

MAP_ANONYMOUS : The mapping is not backed by any file; its contents are initialized to zero. The fd argument is ignored; The offset argument should be zero.

munmap

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

int munmap(void *addr, size_t length);
```

Parameter	Direction	Description
addr	in	Starting address for the new mapping.
length	in	thread attributes. May be NULL, in which case, default attributes are used.

Return Value:
The address of the new mapping is returned as the result of the call.

Description:
The munmap() system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.

The address addr must be a multiple of the page size (but length need not be). All pages containing a part of the indicated range are unmapped, and subsequent references to these pages will generate SIGSEGV. It is not an error if the indicated range does not contain any mapped pages.

Hands on lab:

Open a terminal and write the below program in mmap_shared.c using your preferred editor.

```
$ vi mmap_shared.c
#define SZ (1024*1024)

int main()
{
  int i, fd;
  char *buf;

  if ((fd = open("./1mfile", O_RDWR)) < 0) {
    ...
  }

  buf = mmap(NULL, SZ, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0));
  if ((buf == MAP_FAILED) {
     ...
  }

  for (i=0; i<SZ; i++) {
      buf[i] = 65 + i%26;
  }

  munmap(buf, SZ);
  close(fd);

  return 0;
}
```

Compile and run the program :

```
$ gcc -o mmap_shared mmap_shared.c
$ truncate –s 1G 1mfile
$ ./mmap_shared
```

Memory mapped region in the virtual address space

In the second terminal, run cat /proc/<pid>/maps command. Where <pid> is the above process id.

```
$ cat /proc/`pidof mmap_shared`/maps

00400000-00401000 r-xp 00000000 08:08 19104304 /home/maruthisi/mmap_shared
00600000-00601000 r--p 00000000 08:08 19104304 /home/maruthisi/mmap_shared
00601000-00602000 rw-p 00001000 08:08 19104304 /home/maruthisi/mmap_shared
01208000-01229000 rw-p 00000000 00:00 0        [heap]
7f44ee3ee000-7f44ee5b5000 r-xp 00000000 08:06 101130799 /usr/lib64/libc-
2.25.so
7f44ee5b5000-7f44ee7b5000 ---p 001c7000 08:06 101130799 /usr/lib64/libc-
2.25.so
7f44ee7b5000-7f44ee7b9000 r--p 001c7000 08:06 101130799 /usr/lib64/libc-
2.25.so
7f44ee7b9000-7f44ee7bb000 rw-p 001cb000 08:06 101130799 /usr/lib64/libc-
2.25.so
7f44ee7bb000-7f44ee7bf000 rw-p 00000000 00:00 0
7f44ee7bf000-7f44ee7e6000 r-xp 00000000 08:06 101130590 /usr/lib64/ld-
2.25.so
7f44ee8c4000-7f44ee9c4000 rw-s 00000000 08:08 19104306
/home/maruthisi/1mfile
7f44ee9c4000-7f44ee9c7000 rw-p 00000000 00:00 0
7f44ee9e3000-7f44ee9e5000 rw-p 00000000 00:00 0
7f44ee9e5000-7f44ee9e6000 r--p 00026000 08:06 101130590 /usr/lib64/ld-
2.25.so
7f44ee9e6000-7f44ee9e8000 rw-p 00027000 08:06 101130590 /usr/lib64/ld-
2.25.so
7ffee32f2000-7ffee3313000 rw-p 00000000 00:00 0         [stack]
7ffee33ae000-7ffee33b0000 r--p 00000000 00:00 0         [vvar]
7ffee33b0000-7ffee33b2000 r-xp 00000000 00:00 0         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

In the above output, notice the `rw-s`. The `rw` are due to *prot*, `s` is due to flags. The `19104306` is inode number of the file.

Exercise:

1. Write a program which copies a given file to a new file using mmap’d read and mmap’d write.
2. Write a program which copies a given file to a new file using mmap’d read and systemcall write.