

Lecture 11 — Memory-Mapped I/O

Jeff Zarnett & Mike Cooper-Stachowsky

jzarnett@uwaterloo.ca, mstachowsky@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 5, 2024

Memory-mapped I/O – to communicate with a HW device, an area that looks like main memory really represents the HW device.

This simplifies some operations!

Kind of like how UNIX treats everything as a file...

This contrasts with the older-style **port-mapped** I/O

There are different CPU instructions and separate registers used to communicate with the device.



It's not that this does not work, but it's just more difficult to work with.

Maybe a better name for this is “Memory Mapped Control”.

The device is controlled via setting some voltages (1s, 0s) on the chip.

If we map those to memory locations, we know how to read write the 1s and 0s to the device just as if it were memory.

Hardware devices outside the CPU and memory are sometimes referred to as peripheral devices.

A peripheral is an often optional circuit that interfaces with the CPU.

Examples: UART, I2C, GPIO...

Okay, but where are they?

The data sheet for the device will tell you where the registers are.

One option: absolute address (which is nice).

Other option: base and offset locations.

Why base and offset? Support multiple devices.



An alternative use for memory-mapping in UNIX involves the use of `mmap ()`, a function nominally used to map a file into memory.

Then, interact with the file just like memory!

```
void* mmap( void* address, size_t length, int protection, int flag,  
            int fd, off_t offset );
```

`address`: where you want the mapped region to go; use `NULL`.

`length`: how many bytes to map.

`protection`: rules for how memory can be used.

`flag`: mode for mapping.

`fd`: file descriptor of the file to map.

`offset`: how far from the start of the file mapping begins.

Valid values are `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, and `PROT_EXECUTE`.

They can be combined with the bitwise OR operator.

Whatever flags you choose have to be consistent with how the file was opened with `open`.



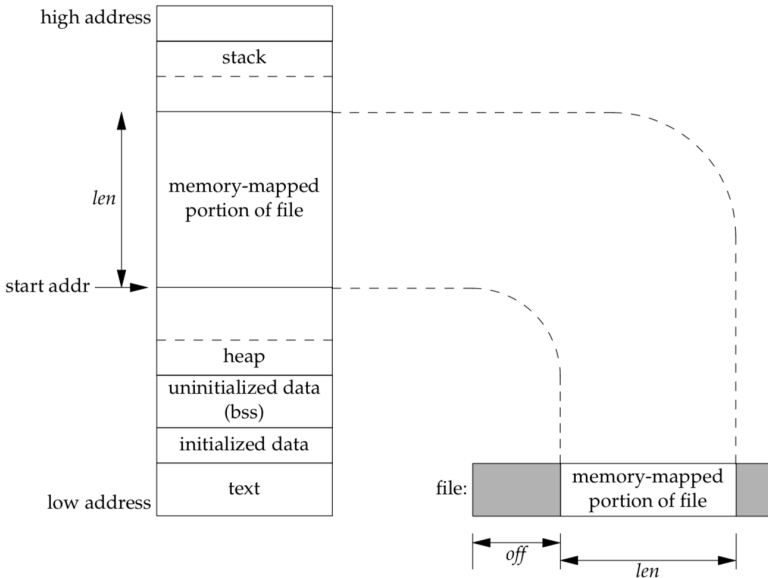
What's the point of PROT_NONE, if all things are forbidden?

Flags can be one of two options: `MAP_PRIVATE` or `MAP_SHARED`.

Private: modifications are not visible to other processes mapping the same file and not written out to the underlying file.

Shared: modifications are visible to other processes and written out to the file... but maybe not instantly.

Memory Mapped File



If we wish to change the protection rules for a section, we use `mprotect`.

```
int mprotect( void* address, size_t length, int prot );
```

`address`: the memory to modify protection of.

`length`: the size of said memory.

`prot`: the new protection rules.



```
int msync( void* address, size_t length, int flags );
```

address: the memory to synchronize.

length: how many bytes to synchronize.

flags: mode for synchronization; use MS_SYNC (blocking).

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

address: the memory to unmap.

length: how many bytes to unmap.

A segment would be unmapped automatically when a process exits, but as always it is polite to unmap it as soon as you know that you are done with it.

Memory Mapping Example

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

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

    int fd = open( "example.txt", O_RDWR );

    struct stat st;
    stat( "example.txt", &st );
    ssize_t size = st.st_size;
    void* mapped = mmap( NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0 );
```

```
int pid = fork();
if ( pid > 0 ) { /* Parent */
    waitpid( pid, NULL, 0 );
    printf("The_new_content_of_the_file_is:_%s.\n", (char*) mapped);
    munmap( mapped, size );
} else if ( pid == 0 ) { /* Child */
    memset( mapped, 0, size ); /* Erase what's there */
    sprintf( mapped, "It_is_now_Overwritten");
    /* Ensure data is synchronized */
    msync( mapped, size, MS_SYNC );
    munmap( mapped, size );
}
close( fd );
return 0;
}
```

The example works acceptably in the sense that we successfully overwrite the data with the new data and the parent process sees the change.

But things get weird if we tried to write fewer bytes than the original message.

In general, the mapped area size cannot change.

Linux has `mremap` but this is not portable...

How does it work, though?

Basic premise: disk blocks mapped to main memory.

First access is a page fault; subsequent accesses are reads/writes.