File System, Part 6: Memory mapped files and Shared memory

Edit New Page

goldcase edited this page 12 days ago · 8 revisions

How does the operating system load my process and libraries into memory?

By mapping the files' contents into the address-space of the process. If many programs only need read-access to the same file (e.g. /bin/bash, the C library) then the same physical memory can be shared between multiple processes.

The same mechanism can be used by programs to directly map files into memory

How do I map a file into memory?

A simple program to map a file into memory is shown below. The key points to notice are:

- mmap requires a filedescriptor, so we need to `open` the file first
- We seek to our desired size and write one byte to ensure that the file is sufficient length
- When finished call munmap to unmap the file from memory.

This example also shows the preprocessor constants "**LINE**" and "**FILE**" that hold the current line number and filename of the file currently being compiled.

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


int fail(char *filename, int linenumber) {
  fprintf(stderr, "%s:%d %s\n", filename, linenumber, strerror(errno));
  exit(1);
  return 0; /*Make compiler happy */
}
#define QUIT fail(__FILE__, __LINE__ )

int main() {
  // We want a file big enough to hold 10 integers
  int size = sizeof(int) * 10;

  int fd = open("data", O_RDWR | O_CREAT | O_TRUNC, 0600); //6 = read+write for m
```

```
  lseek(fd, size, SEEK_SET);
  write(fd, "A", 1);

  void *addr = mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  printf("Mapped at %p\n", addr);
  if (addr == (void*) -1 ) QUIT;

  int *array = addr;
  array[0] = 0x12345678;
  array[1] = 0xdeadc0de;

  munmap(addr,size);
  return 0;

}
```

The contents of our binary file can be listed using hexdump

```
$ hexdump data
0000000 78 56 34 12 de c0 ad de 00 00 00 00 00 00 00 00
0000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000020 00 00 00 00 00 00 00 00 41
```

The careful reader may notice that our integers were written in least-significant-byte format (because that is the endianess of the CPU) and that we allocated a file that is one byte too many!

The `PROT_READ | PROT_WRITE` options specify the virtual memory protection. The option `PROT_EXEC` (not used here) can be set to allow CPU execution of instructions in memory (e.g. this would be useful if you mapped an executable or library).

What are the advantages of memory mapping a file

For many applications the main advantages are:
Simplified coding - the file data is immediately available. No need to parse the incoming data and store it in new memory structures.
Sharing of files - memory mapped files are particularly efficient when the same data is shared between multiple processes.

Note for simple sequential processing memory mapped files are not necessarily faster than standard 'stream-based' approaches of `read` / fscanf etc.

How do I share memory between a parent and child process?

Easy - Use `mmap` without a file - just specify the MAP_ANONYMOUS and MAP_SHARED options!

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
```

```c
#include <sys/stat.h>
#include <sys/mman.h> /* mmap() is defined in this header */
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main() {

  int size = 100 * sizeof(int);
  void *addr = mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS,
  printf("Mapped at %p\n", addr);

  int *shared = addr;
  pid_t mychild = fork();
  if (mychild > 0) {
    shared[0] = 10;
    shared[1] = 20;
  } else {
    sleep(1); // We will talk about synchronization later
    printf("%d\n", shared[1] + shared[0]);
  }

  munmap(addr,size);
  return 0;
}
```

Can I use shared memory for IPC ?

Yes! As a simple example you could reserve just a few bytes and change the value in shared memory when you want the child process to quit. Sharing memory is a very efficient form of inter-process communication because there is no copying overhead - the two processes literally share the same *physical* frame of memory.

Go to File System: Part 7