

Inter-Process Communication: Shared Memory



CMPT 201

Slides 11.2

Based on content
created by Dr. Steve Ko
© Dr. B. Fraser

Topics

- Since memory is so useful and easy to access, **can we load a whole file into memory?**
- If processes have separate memory spaces, how can **two processes share memory?**

Memory Mapping

Intro to Memory Mapping

- Memory mapping
 - It's not *just* for IPC, but we'll need it!
- Uses for Memory Mapping:
 - ..
vs using `read()/write()`
 - Allocating memory
 - ..
(useful for embedded systems; shared between processors!)

mmap()

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

..

- **addr**: starting address of the new mapping.
*Usually **NULL** so OS pick the address.*
 - **length**: # bytes in mapping.
 - **prot**: Memory protection for executable, readable, writable, or not accessible.
 - **flags**: **MAP_SHARED** or **MAP_PRIVATE**, and optionally **MAP_ANONYMOUS**. (explained below)
 - **fd**: .. (explained below)
 - **offset**: the offset into the file to be mapped.
- Returns a pointer to the beginning of the new mapping.

Types of Memory Mapping

- Two types of memory mappings

- ..

- File is loaded into a memory region
 - File I/O becomes memory access:
 - Replace `read()/write()` calls with pointer access to read or write file.
 - This is called a.. `memory-mapped file`.
 - `flag` argument: `MAP_ANONYMOUS` flag is **not** set.

- ..

- This is another way to allocate memory to our process (in addition to `sbrk()`).
 - `malloc()` uses both `sbrk()` and `mmap()`.
 - `flag` argument: `MAP_ANONYMOUS` flag is set.

Shared vs Private

- Memory Mapping can be shared or private.
- Shared Mapping:
 - ..
 - E.g., ..
 - Since memory is cloned, the parent and the child will share the same mapping.
 - Or, multiple processes can map the same file.
- Private Mapping:
 - Changes in one process's memory mapping
 - ..

4 Possibilities

- **Private file mapping:**

- A **file** is mapped to a process as a **private** mapping.
- ..

- **Shared file mapping:**

- A **file** is mapped to a process as a **shared** mapping.
- Changes propagate to:
 - ..
 - **and other processes** mapping same file.

- **Private anonymous mapping:**

- **More memory is allocated** to the calling process.
- ..

(changes not shared).

- **Shared anonymous mapping:**

- **More memory is allocated** to the calling process.
- Memory is **shared**; changes propagate to other process!

mmap() arguments: offset = 0
fd = -1 or **shm_open()**
flag |= **MAP_ANONYMOUS**

Unmap

- `int munmap(void *addr, size_t length);`
 - Unmaps the mapped memory.

ABCD: Memory Mapping

- Which of the options below is best described by:
 - Used to allow fast access to a temporary copy of a file.
 - Used to have two processes access the same memory so they can both access a shared data structure.
 - Used to allow any number of processes to edit a file and see each others edits, plus reflect changes to file on disk.

- a) Shared anonymous mapping
- b) Private anonymous mapping
- c) Shared file mapping
- d) Private file mapping

Memory Mapping Activity

- **Activity:** memory-mapped file I/O.
 - Modify the example from `man mmap` as follows:
 - Receive only **one command-line argument**, which is a **file name**.
 - Create a **file memory mapping** for the entire file.
 - **Print out the content** of the entire memory mapping.

Shared Memory

Sharing memory

Two different ways to share memory between processes.

- For Related processes:

..

- `mmap()` with `MAP_SHARED | MAP_ANONYMOUS`
(i.e., shared anonymous)

- For Unrelated Processes:

..

- `man 7 shm_overview`
 - `shm_open()`: Open a shared memory object
 - `ftruncate()`: Set size
 - `mmap()`: Create memory mapping

shm_open()

```
int shm_open(const char *name, int oflag, mode_t mode)
```

- Similar to opening a file, but it's shared memory.
 - Just like creating a file; listed in `/dev/shm/`
 - E.g., `ls /dev/shm/somename`
- **Returns:** file descriptor for..
- **name:** Known by all participating processes.
General form: `/somename`.
- **flag:** `O_CREAT` flag set when creating a new object.
- **mode:** For permissions on creation.

Size and Map

```
int ftruncate(int fd, off_t length)
```

- Memory object is created with size 0.
- `ftruncate()` sets its size.

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

- Create memory map for memory object
(after created by `shm_open()` and size set with `ftruncate()`).
- ..
(from `shm_open()`).

Cleanup

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

- Unmap shared memory when no longer needed.

```
int shm_unlink(const char *name)
```

- ..
when done with shared memory.
 - Removes file from `/dev/shm/`.
- However, processes still using the shared memory object keep using it.

ABCD: shm_open()

- When do we need to call `shm_open()`?
 - a) When two processes want to share memory.
 - b) When a parent and child processes want to share memory without calling `fork()`.
 - c) When two unrelated processes want to share memory.
 - d) When two processes share access to a file and each process knows the file's name.

Activity: Shared Memory

- **Activity**
 - Write **two programs** that communicates with each other **via shared memory**.
 - They should each **receive a shared memory object file name** as the only **command-line** argument.
 - One program should **write an integer** to the shared memory
 - The other program should **read the integer** written by the first program from the shared memory.

Summary

- Two processes can communicate by sharing memory.
- `mmap()`
 - Creates a **memory mapping** of a **file** or **some memory**.
 - Usually copied by `fork()`
 - Useful for **parent-child** shared memory.
 - `mmap()`, `munmap()`
- `shm_open()`
 - Creates a **named shared memory object**.
 - Useful for **unrelated processes** to share memory.
 - `shm_open()`, `ftruncate()`, `mmap()`, `munmap()`, `shm_unlink()`