Inter-Process Communication: Shared Memory

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

Topics

- Since memory is so useful and easy to access, can we load a whole file into memory?
  - But without incurring immediate I/O overhead?
- 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*
  - Loading a file into memory
    - Without using `read()/write()`
  - Allocating memory
  - Accessing memory-mapped devices via `/dev/mem`
    - Useful for embedded systems; shared between processors!

mmap()

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

- *mmap() creates a **memory mapping***
  - **addr:** starting address of the new mapping (usually NULL, kernel picks the address)
  - **length:** number of bytes mapped
  - **prot:** memory protection, i.e. read/write/execute
  - **flags:** MAP_SHARED or MAP_PRIVATE, and optionally MAP_ANONYMOUS (explained later)
  - **fd:** the open file to be mapped (explained later)
  - **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 mapping**
    - 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 **cleared,** i.e. **not** set
  - **Anonymous mapping**
    - 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

- *Shared Mapping*
  - Multiple processes can share a mapping
    - E.g., create a mapping (file or anonymous) and `fork()` a child
    - Since process state 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 do not appear for other processes, and are not written to a file

Four Mapping Types

	File	***Anonymous***

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

Private	**Private file mapping** A file is mapped to a process as a private mapping. **Changes are not written to the file or shared with other processes.**	**Private anonymous mapping** More memory is allocated to the calling process. `fork()` copies memory, but each process has a private copy. **Changes are not shared.**
Shared	**Shared file mapping** A file is mapped to a process as a shared mapping. **Changes propagate to:** • **the real file** • **other processes mapping same file**	**Shared anonymous mapping** More memory is allocated to the calling process. **Memory is shared, changes propagate to other process!**

Unmap

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

- Unmaps the mapped memory

Audience Participation - Memory Mapping

- Which option is best described by the following:
  1) Gives fast access to a temporary copy of a file
  2) Gives two processes access to the same memory so they can both use a shared data structure
  3) Allows 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

Activity - Memory Mapping

- Activity: modify the example from `man mmap` as follows (25m)
  - 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

Shared Memory

- *Sharing memory between related processes*
  - Create memory map then share with `fork()`
  - `mmap()` with `MAP_SHARED | MAP_ANONYMOUS` (i.e., shared anonymous)
- *Sharing memory between **unrelated** processes*
  - Open a shared memory object and then `mmap()`
  - `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 shared memory
  - Transient, not persistent
  - There will be a file-like object in `/dev/shm/`
    - Try: `ls /dev/shm/somename`
  - Returns a file descriptor for a shared memory object
  - `name`: Should be 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)
```

- Use ftruncate() to set size of memory object
  - Memory object is created with size 0, so this is important

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

- Create memory map for memory object
  - (after it's been created by `shm_open()`, and the size set with `ftruncate()`).
  - Pass shared memory object file descriptor as fd (from `shm_open()`)

Cleanup

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

- Unmap shared memory when no longer needed

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

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

Audience Participation - 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 (25m)
  - 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()`