

CSE 220: Systems Programming

Memory and Concurrency

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

Memory and Concurrency

We have discussed [shared state](#) and concurrency.

However, [the issues go deeper than that!](#)

Shared state is in [shared memory](#).

Memory has some [confusing properties](#) when it is shared.

How does memory [become](#) shared, anyway?

Types of Shared Memory

There are several “types” of shared memory in concurrent programming:

- Memory used by the same thread in the same process at different times (and maybe asynchronously)
- Memory used by different threads in the same process (maybe at the same time)
- Memory used by different processes (maybe at the same time)

The first is mostly non-problematic.

The second two **require a little extra work**.

Acquiring Shared Memory

Memory shared **within a process** requires no special setup.

Sharing memory **between processes** requires kernel assistance.

There are several methods for creating shared memory:

- Creating a shared mapping within a process **before forking**¹
- Attaching to a **named mapping** with `shm_open()`
- Attaching to a **memory-mapped file**

We will explore only the latter.

¹`fork()` is the POSIX method for creating a new process.

Sharing Memory

So far, we have explored only one way to share memory:
Threads within a process share **all memory**.

It is often useful to **share memory in a controlled way**.

For example:

- A typed data structure (such as a list or tree)
- A buffer of raw bytes
- A synchronization tool (such as a producer-consumer queue)
- ...

Implicitly Shared Memory

Processes have a lot of **implicitly shared memory**:

- Shared libraries
- Executable images
- Kernel memory
- ...

This memory is **not obviously shared**, however.

It is either **read-only** or **hidden**.

Explicitly Sharing Memory

A process can **request explicit memory sharing**.

That memory may be **mutable** and changes may be visible **between processes**.

Like all other resources, the **kernel sets up shared memory**.

POSIX systems offer two fundamental **system calls** (and three methods) for explicitly sharing memory:

- `mmap()` **maps a file** into memory, and **changes to the file** can be shared between processes
- `mmap()` can also be used to **create an anonymous shared mapping** shared between parent and child processes
- `shm_open()` **opens a named shared memory region**

mmap()

You have **already seen** `mmap()`.

It asks the kernel to **manipulate the process virtual memory map**.

Its analogue is `munmap()`.

It is **quite complicated** to use properly.

The original use of `mmap()` was to **map a file into memory**.

Using `mmap()`

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

The flags argument determines **what kind of mapping** is created.

It **must** include either `MAP_PRIVATE` or `MAP_SHARED`.

It may include **many other options**.

`MAP_ANONYMOUS`, in particular, means **do not map a file**.

Using `mmap()`

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

The `prot` argument determines the **permissions** of the mapping. It must be either `PROT_NONE` or a bitwise-or of:

- `PROT_READ`: the mapping is readable
- `PROT_WRITE`: the mapping is writeable
- `PROT_EXEC`: the mapping is **executable**

The selected protection **must match the open fd mode**.
(*E.g.*, an `O_RDONLY` file cannot be mapped `PROT_WRITE`.)

Example of `mmap()`

From your malloc project:

```
void *mapping = mmap(NULL, size,
                    PROT_READ | PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS,
                    -1, 0);
```

- NULL addr because we don't care
- readable, writeable mapping
- MAP_PRIVATE so the map is not shared, and MAP_ANONYMOUS because there's no file
- fd is -1 because there's no file
- The size is as requested, with no offset

Shared Mapping with `mmap()`

```
void *mapping = mmap(NULL, size,  
                    PROT_READ | PROT_WRITE,  
                    MAP_SHARED | MAP_ANONYMOUS,  
                    -1, 0);
```

This mapping will be **preserved across `fork()`**.

The memory will be **at the same address** in both processes.

POSIX semaphores created in the shared memory, or created in other memory, with `pshared = 1`, will synchronize processes.

Mapping a File

```
int fd = open("somefile", O_RDWR);
void *mapping = mmap(NULL, 4096,
                    PROT_READ | PROT_WRITE,
                    MAP_SHARED, fd, 0);

close(fd);
```

(Similar to the pixmaps lab.)

This maps the first 4 KB of somefile to the address mapping.

This mapping will be **with shared with all processes** mapping the same location in the same file.

Note that **the file can be deleted** after it is mapped.

Executable Loading with `mmap()`

Recall that **executables on disk** are mapped into memory.

This is accomplished **using `mmap()`**.

The various **ELF sections** are mapped appropriately:

- `.text` with `PROT_READ | PROT_EXEC`
- `.rodata` with `PROT_READ`
- `.data` with `PROT_READ | PROT_WRITE`

The text and read-only data can be **`MAP_SHARED` to save RAM**.

The BSS is an **anonymous mapping** using `MAP_ANONYMOUS` and `PROT_READ | PROT_WRITE`.

Shared Memory Without a File

Mapping a file to share memory **can be convenient**:

- It persists when **no process is using it**
- It persists **between reboots**
- It can be **easily analyzed** with standard utilities

Mappings can also be created **without a file**.

The `shm_open()` system call creates a **file descriptor** referencing a **kernel memory buffer**.

The file descriptor returned by `shm_open()` **is usable with `mmap()`**.

Consistency

Many problems with **memory and concurrency** are with **consistency**.

Within the **dedicated computer** model, we have expectations:

- Writing to a memory location is **immediate**
- Writes to a memory location are **durable**

With **concurrent flows**, these expectations can break.

We have already seen how to mitigate this with **synchronization**.

However, **synchronization must control more than timing**.

Temporal Synchronization

Up to now, we have thought of synchronization as a **temporal construction**:

- Operation o_1 occurs before operation o_2
- A sequence of operations is not interrupted

However, there are also **spatial concerns**.

- An operation **is visible to another part of the system**.

Caching

Modern computers have **many layers of caching**.

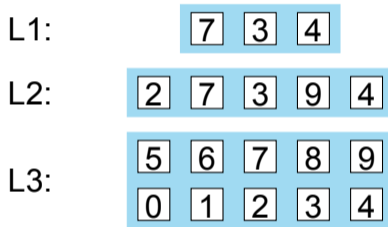
Some of these caches are **shared**, some are **local**:

- Local to a particular **CPU core** (*E.g.*, L1)
- Local to a **subset of cores** (*E.g.*, L3)
- Local to a **process**
- ...

Writes to a **local cache** may not be **visible** to concurrent flows.

Caching Structure

Each level of cache stores **blocks** from the next level.



Reads come from the **first level with the desired data**.

Writes **eventually** propagate to all levels.

Write Propagation

The consistency problem comes from that **eventually**:
Writes **eventually** propagate to all levels.

If a cache is **local** to a core or set of cores, reads from **other cores** will not reflect its contents.

Consider **registers**: only **one core** sees them!

As previously discussed:

- Many operations (even **instructions**) have multiple steps
- Some of those steps are performed **in registers**

Write Propagation Problem

Consider:

- Core C_0 executes a write for memory location m
- The write is stored to C_0 's L1 cache
- Core C_1 executes a read for memory location m
- The location m is not in C_1 's L1 or L2
- C_1 reads m from shared L3
- C_0 's L1 propagates m to C_0 's L2
- C_0 's L2 propagates m to the shared L3

Write Propagation II

Temporal synchronization can guarantee that a register is written to memory.

To guarantee it isn't cached, we need memory barriers.

A memory barrier does one or more of:

- Blocks the current core until a write is visible to all cores
- Blocks the current core until all writes are visible
- Blocks all cores from accessing a location until a write is visible
- ...

Memory Barriers

Memory barriers are sometimes called memory fences².

Memory barriers are hardware functions.

Most processors have barrier instructions.

For example:

- mfence on x86-64
- dmb on ARM
- many atomic instructions

²A memory barrier and a synchronization barrier are not the same thing.

Write Propagation with Barriers

Consider:

- Core C_0 executes a write for memory location m
- The write is stored to C_0 's L1 cache
- Core C_1 issues a barrier for all writes to m
- Core C_1 executes a read for memory location m
- Core C_1 blocks because C_0 is writing m
- C_0 's L1 propagates m to C_0 's L2
- C_0 's L2 propagates m to the shared L3
- C_1 reads m from shared L3

Synchronization and Barriers

Synchronization primitives **use memory barriers**.

These functions, for example, all have barriers:

- `fork()`
- `pthread_mutex_lock()`
- `pthread_mutex_unlock()`
- `pthread_create()`
- `pthread_join()`
- ...

Basically **all of the POSIX synchronization functions**.

C and Memory Barriers

The C language makes **very few guarantees** regarding barriers.

C11 has some fence (barrier) operations.

C99 **does not expose** barriers.

In general **libraries or OS functions** (such as Pthreads) are required for **thread-safe operation in C**.

Some C compilers may provide barriers (e.g., `__builtin_ia32_mfence()` in GCC).

Summary

- Caching and CPU architecture require **more than just temporal synchronization**
- **Memory barriers** force data visibility **across cores**
- Memory barriers are a **hardware feature**
- POSIX synchronization primitives **use memory barriers**
- Shared memory **requires kernel assistance**

References I

Required Readings

- [1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 6: Intro, 6.3; Chapter 9: 9.8. Pearson, 2016.

License

Copyright 2018, 2020 Ethan Blanton, All Rights Reserved.
Copyright 2019 Karthik Dantu, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see <https://www.cse.buffalo.edu/~eblanton/>.