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 not explore these methods further.

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

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
- Local to a subset of cores
- Local to a process
- …

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

Why Cache?

Caches are used for performance reasons, in levels:

Level	*Type*	*Size*	*Access Time*
L0	CPU registers	O(100 B)	~0 clock cycles
L1	Level 1 cache	O(10 KB)	~1-5 clock cycles
L2	Level 2 cache	O(100 KB)	~10+ clock cycles
L3	Level 3 cache	O(1 MB)	~30+ clock cycles
L4	Main memory	O(10 GB)	~100+ clock cycles

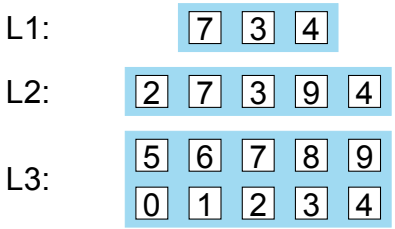
Lower levels are much faster but much smaller.

L0-L1 are often local to a core, L2-3 to a core or subset of cores. L4 is typically shared.[2]

[2] Architectures where it is not are called NUMA.

Caching Structure

Each level of cache stores blocks from the next level.



Block location and size may vary from level to 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
- Prevents CPU instruction reordering from affecting this instruction
- …

Memory Barriers

Memory barriers are sometimes called memory fences[3].

Memory barriers are hardware functions.

Most processors have barrier instructions.

For example:

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

[3] A 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
- Caches are much faster than main RAM
- 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

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