Advanced Multi-Thread Programming

Yifan ZHU <i@zhuyi.fan>

October 20, 2021

Memory Model and Barriers

- Memory Architecture
- Barriers

More on Atomic Variables

- Half Barriers
- Architecture Consideration
- Relaxed Memory Order

Thread in Deep

- How is Thread Created in Linux?
- How could a Thread Identify Itself?

Mutex in Deep

- Spin Lock
- Futex
- TSX/HLE
- Locking Stages

Illusion of CPU Architecture

Figure: Illusion of Multi-core CPU

Hierarchical CPU Architecture

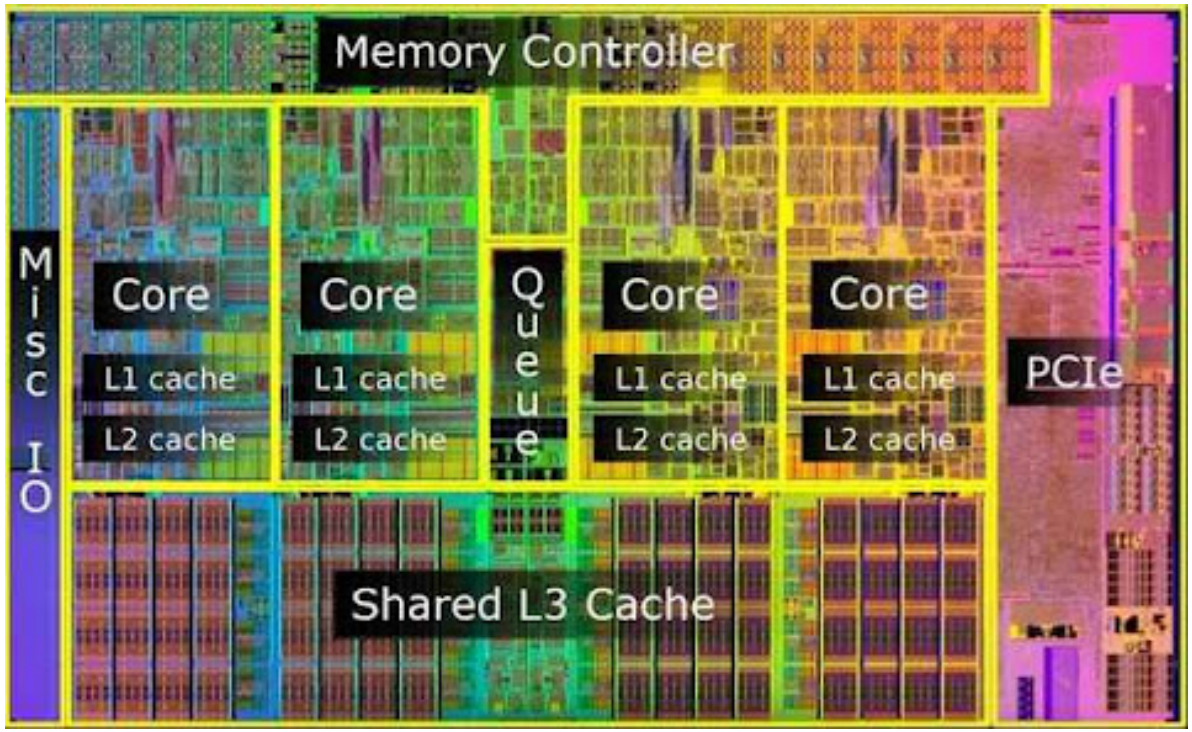

Figure: Intel Die and Cache Diagram

Hierarchical CPU Architecture

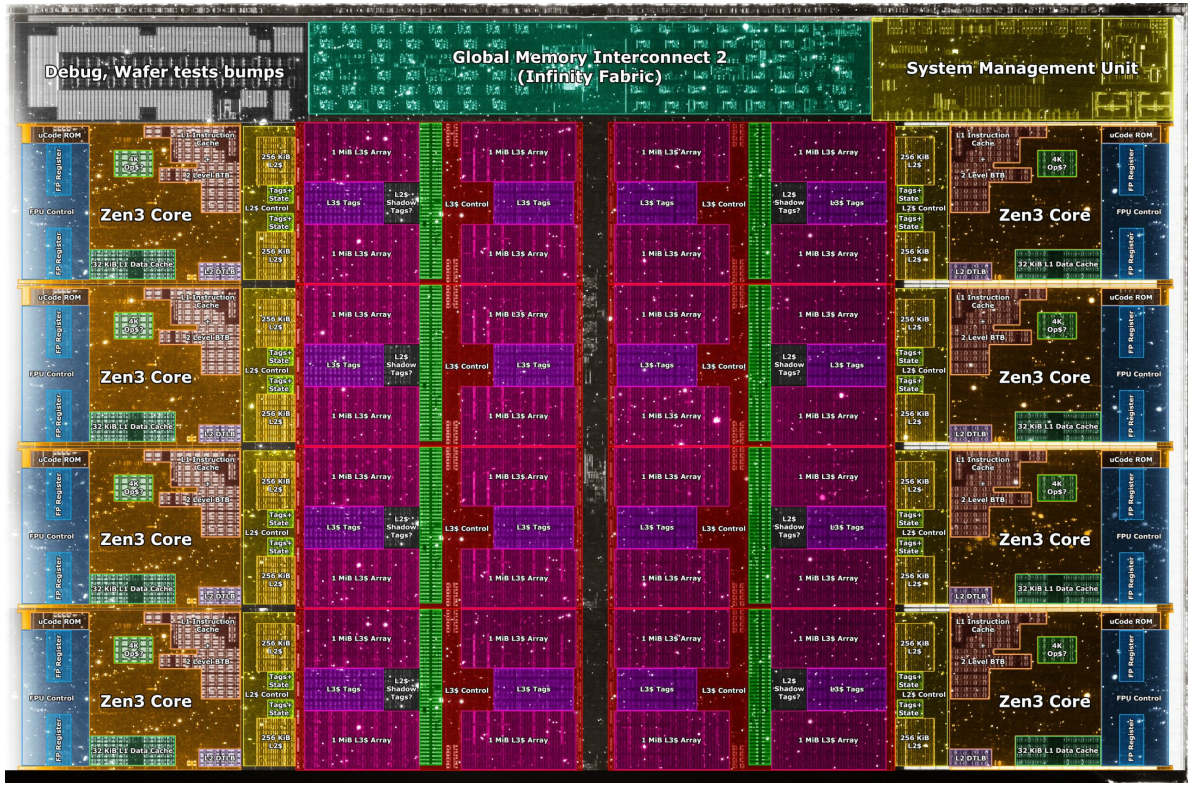

Figure: Ryzen Dies and Cache Diagram

NUMA Architecture

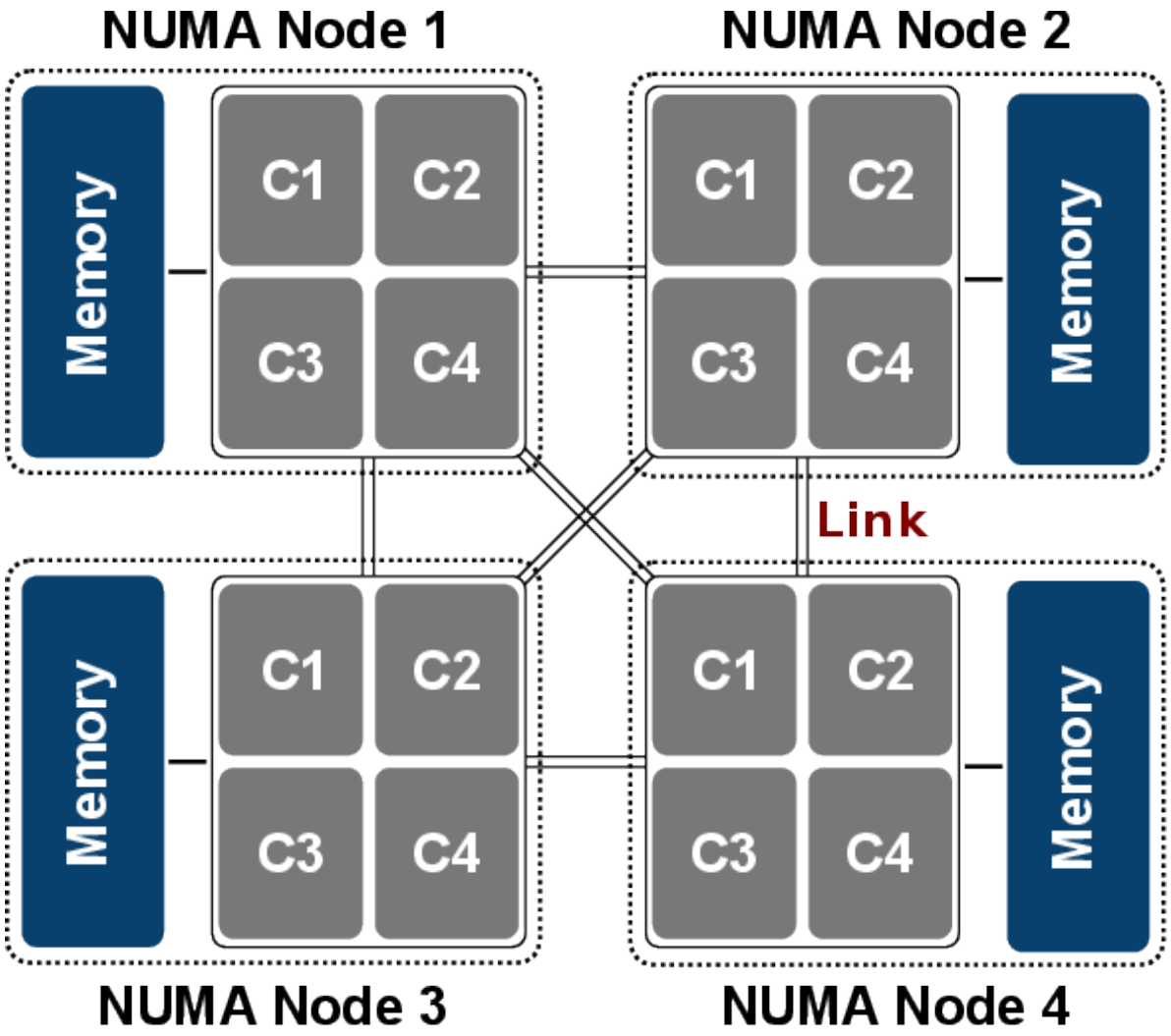

Figure: NUMA Architecture and Memory Interconnections

Write Buffers

Memory operations can be really costly, hence caches and write buffers are essential for high performance

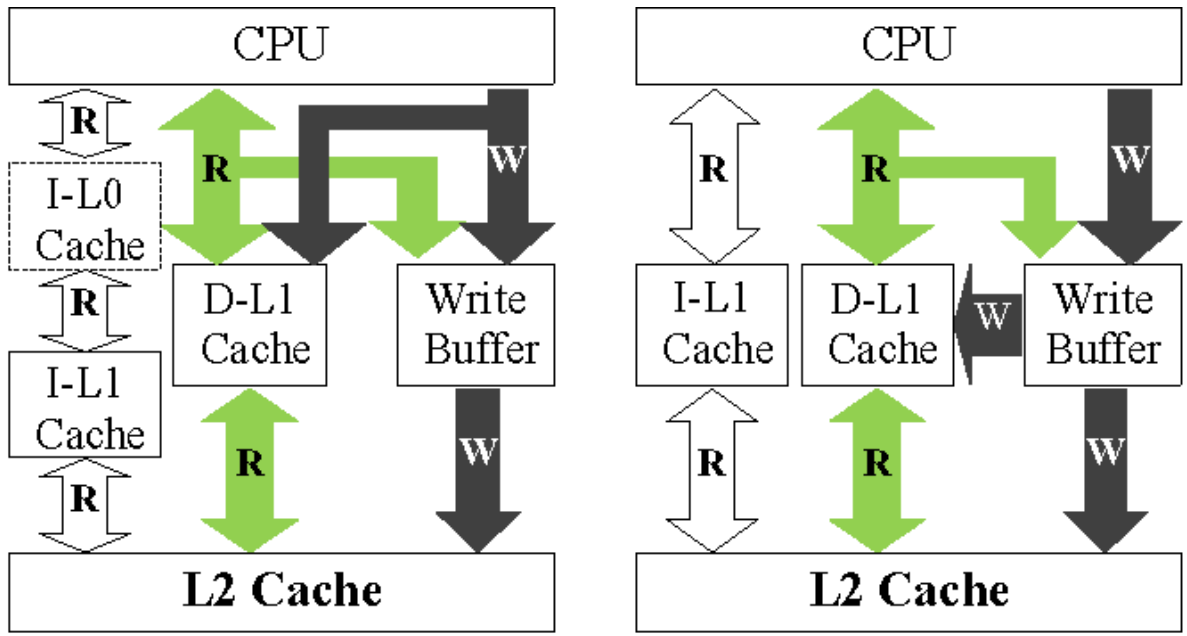

Figure 1. Memory operations in the conventional 2-level (3-

Figure: Writer Buffers

Memory Barriers

The core idea is to maintain the illusion of consistent memory.

Key Concepts

- Sequential Consistency: the code is executed as you wished.
- Data Races: two threads access the same data location with at least one of them being a writer.

Why Sequential Consistency Can Be an Issue?

- Compiler: reorder instructions to optimize your code
- CPU: Launch multiple instructions in one go
- Cache: Multiple IO requests can be issued in the same time

Why Sequential Consistency Can Be an Issue?

- Compiler: reorder instructions to optimize your code
- CPU: Launch multiple instructions in one go
- Cache: Multiple IO requests can be issued in the same time

Dekker's Algorithm

```
// Thread 1
flag1 = 1;  // a
if (flag2 != 0)  // b
{ /* do something else */ }
else
{ /* critical section */ }
// Thread 2
flag2 = 1; // c
if (flag1 != 0)  // d
{ /* do something else */ }
else
{ /* critical section */ }
```

Compiler Fences

```
static void
fast_path(int *read_b)
{
    a = 1;
    asm volatile ("" : : : "memory");
    *read_b = b;
}
static void
slow_path(int *read_a)
{
    b = 1;
    membarrier(MEMBARRIER_CMD_GLOBAL, 0, 0);
    *read_a = a;
}
```

CPU Barriers

```
CPU MEMORY BARRIERS
-------------------

The Linux kernel has eight basic CPU memory barriers:

        TYPE             MANDATORY                SMP CONDITIONAL
        ================ ======================== ============================
        GENERAL          mb()                     smp_mb()
        WRITE            wmb()                    smp_wmb()
        READ             rmb()                    smp_rmb()
        DATA DEPENDENCY                           READ_ONCE()
```

Figure: Linux Barriers

Write Barriers

Write (or store) memory barriers.
A write memory barrier gives a guarantee that all the STORE operations specified before the barrier will appear to happen before all the STORE operations specified after the barrier with respect to the other components of the system.

Read Barriers

Read (or load) memory barriers.
A read barrier is a data dependency barrier plus a guarantee that all the LOAD operations specified before the barrier will appear to happen before all the LOAD operations specified after the barrier with respect to the other components of the system.

Data Dependency Barriers

A data dependency barrier is a weaker form of read barrier. In the case where two loads are performed such that the second depends on the result of the first (eg: the first load retrieves the address to which the second load will be directed), a data dependency barrier would be required to make sure that the target of the second load is updated after the address obtained by the first load is accessed.

General Barriers

A general memory barrier gives a guarantee that all the LOAD and STORE operations specified before the barrier will appear to happen before all the LOAD and STORE operations specified after the barrier with respect to the other components of the system.

SC-DRF

Sequential Consistent for Data Race Free Programs

Transaction Model

```
int tmp = x.load(); // acquire, transaction begins
tmp = tmp + 1;
x.store(tmp); // release, transaction ends
```

Transaction Model (Mutex)

```
{
    lock_guard<mutex> guard;
    // do something
}
```

Transaction Model (Mutex)

How does compile know the where to stop reordering?

- mutex is normally implemented with compiler intrinsic.
- compiler can treat opaque functions as full barriers.

Transaction Model (Atomic)

```
while (permitted != my_own) { /* spin */ }
 // do something
permitted = someone_else;
```

Transaction Model (STM/HTM)

```
synchronize {
    // do something
}

// HLE
xbegin();
// do something;
xend();
```

Half Barrier

- Move Into but not move out of.
- No cross over of any pair of Acquire/Release

Half Barrier

Half VS Full

- Portability
- Easiness
- Performance

Codegen

	load	load(a)	store	store(a)	cas
x86_64	mov	mov	mov	xchg	cmpxchg(16b)
AARCH64	ldr	ldra	str	strl	....
ARMv7	ldr	ldr;dmb	str	dmb; str; dmb	....

Why x86 can use normal mov for atomic load?

- No reordering of read-read
- No reordering of write-write
- No reordering of write with older read
- No reordering read/write with lock instruction (xchg).

Relaxed Memory Order

```
// no acqurie/release sematic; only atomicity
std::memory_order_relaxed
```

How is Thread Created in Linux?

Name

clone, __clone2 - create a child process

Synopsis

```
#define _GNU_SOURCE             /* See feature_test_macros(7) */
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

Figure: SYS_clone

Pthread Self

```
static inline struct pthread *__pthread_self()
{
    struct pthread *self;
    __asm__ __volatile__ ("mov %%fs:0,%0" : "=r"
↪   (self) );
    return self;
}
```

Pthread Self

```
#include <asm/prctl.h>          /* Definition of ARCH_* constants */
#include <sys/syscall.h>        /* Definition of SYS_* constants */
#include <unistd.h>

int syscall(SYS_arch_prctl, int code, unsigned long addr);
int syscall(SYS_arch_prctl, int code, unsigned long *addr);
```

Figure: Setting Register

Pthread Self

```
struct pthread {
        /* Part 1 -- these fields may be external or
         * internal (accessed via asm) ABI. Do not change. */
        struct pthread *self;
#ifndef TLS_ABOVE_TP
        uintptr_t *dtv;
#endif
        struct pthread *prev, *next; /* non-ABI */
        uintptr_t sysinfo;
#ifndef TLS_ABOVE_TP
#ifdef CANARY_PAD
        uintptr_t canary_pad;
#endif
        uintptr_t canary;
#endif
```

Figure: Special Handle of Base Register

Spin Lock

```
struct Lock {
    std::atomic<bool> lock_ = {false};
    void lock() { while(lock_.exchange(true)); }
    void unlock() { lock_.store(false); }
};
```

Spin Lock (Explicit Memory Order)

```
struct tas_lock {
std::atomic<bool> lock_ = {false};
void lock() { while(lock_.exchange(true,
 ↪  std::memory_order_acquire)); }
void unlock() { lock_.store(false,
 ↪  std::memory_order_release); }
};
```

Spin Lock (Reduce Cache Sync)

```
struct Lock {
    ...
    void lock() {
        for (;;) {
            if (!lock_.exchange(true,
↪   std::memory_order_acquire)) {
                break;
            }
            while
↪   (lock_.load(std::memory_order_relaxed));
        }
    }
    ...
};
```

Spin Lock (CPU Relax)

```
    ...
    void lock() {
        for (;;) {
            if (!lock_.exchange(true,
↪   std::memory_order_acquire)) {
                break;
            }
            while
↪   (lock_.load(std::memory_order_relaxed)) {
                __builtin_ia32_pause();
            }
        }
    }
    ...
};
```

Futex

```
private:
// 0 means unlocked
// 1 means locked, no waiters
// 2 means locked, there are waiters in lock()
    std::atomic<int> atom_;
```

Futex

```
void lock() {
    int c = cmpxchg(&atom_, 0, 1);
    // If the lock was previously unlocked, there's nothing else for us to do.
    // Otherwise, we'll probably have to wait.
    if (c != 0) {
        do {
            // If the mutex is locked, we signal that we're waiting by setting the
            // atom to 2. A shortcut checks is it's 2 already and avoids the atomic
            // operation in this case.
            if (c == 2 || cmpxchg(&atom_, 1, 2) != 0) {
                // Here we have to actually sleep, because the mutex is actually
                // locked. Note that it's not necessary to loop around this syscall;
                // a spurious wakeup will do no harm since we only exit the do...while
                // loop when atom_ is indeed 0.
                syscall(SYS_futex, (int*)&atom_, FUTEX_WAIT, 2, 0, 0, 0);
            }
            // We're here when either:
            // (a) the mutex was in fact unlocked (by an intervening thread).
            // (b) we slept waiting for the atom and were awoken.
            //
            // So we try to lock the atom again. We set teh state to 2 because we
            // can't be certain there's no other thread at this exact point. So we
            // prefer to err on the safe side.
        } while ((c = cmpxchg(&atom_, 0, 2)) != 0);
    }
}
```

Futex

```
void unlock() {
    if (atom_.fetch_sub(1) != 1) {
        atom_.store(0);
        syscall(SYS_futex, (int*)&atom_, FUTEX_WAKE,
↪   1, 0, 0, 0);
    }
}
```

HLE

```
void elided_lock_wrapper(lock) {
    if (_xbegin() == _XBEGIN_STARTED) {    /* Start transaction */
        if (lock is free)    /* Check lock and put into read-set */
        return;         /* Execute lock region in transaction */

        _xabort(0xff);         /* Abort transaction as lock is busy */
    }                 /* Abort comes here */
    take fallback lock
}

void elided_unlock_wrapper(lock) {
    if (lock is free)
    _xend();         /* Commit transaction */
    else
    unlock lock;
}
```

Schedule Yield

If the calling thread is the only thread in the highest priority list at that time, it will continue to run after a call to **sched_yield**().

POSIX systems on which **sched_yield**() is available define **_POSIX_PRIORITY_SCHEDULING** in *<unistd.h>*.

Strategic calls to **sched_yield**() can improve performance by giving other threads or processes a chance to run when (heavily) contended resources (e.g., mutexes) have been released by the caller. Avoid calling **sched_yield**() unnecessarily or inappropriately (e.g., when resources needed by other schedulable threads are still held by the caller), since doing so will result in unnecessary context switches, which will degrade system performance.

sched_yield() is intended for use with real-time scheduling policies (i.e., **SCHED_FIFO** or **SCHED_RR**). Use of **sched_yield**() with nondeterministic scheduling policies such as **SCHED_OTHER** is unspecified and very likely means your application design is broken.

Figure: Sched Yielding

Mutli-Stage Mutex

1. try HLE first
2. spin several rounds
3. (optional) the previous step can be combined with yield (one can consider exponential backoff algorithm)
4. fallback to futex (slow path)