CS60203: Design Optimization of Computing Systems *September 2, 2024*

Multicore Programming

Instructor: Mainack Mondal *Scribe-By: Manaswi Raj*

Introduction

This document explores various aspects of multicore programming, including the fundamentals of simultaneous multithreading, symmetric multiprocessors, Non-Uniform Memory Access (NUMA), and challenges in multiprogramming.

Simultaneous Multithreading (SMT)

Simultaneous Multithreading is based on the idea that a single thread cannot fully utilize the throughput of a CPU core. Hyperthreading creates two logical cores from a single physical core, improving utilization.

Why is SMT disabled in High-Frequency Trading (HFT)? HFT systems prioritize low latency and deterministic behavior. SMT introduces non-determinism and potential inconsistencies due to thread behavior, making latency unpredictable. Additionally, running multiple threads can cause context-switching overheads.

Symmetric Multiprocessor (SMP)

In an SMP system, some hardware resources are exclusive to each core (e.g., registers), while others are shared among all cores (e.g., lower-level caches and memory).

Challenges: The shared instruction and data memory can become bottlenecks, limiting scalability.

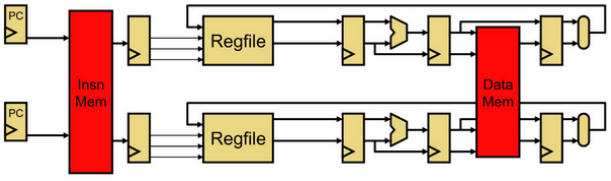


Figure 1: Symmetric Multiprocessor Design

Non-Uniform Memory Access (NUMA)

NUMA architecture is used when memory access costs are non-uniform, meaning accessing different memory locations can have different latencies. This is particularly useful in systems with large memory requirements.

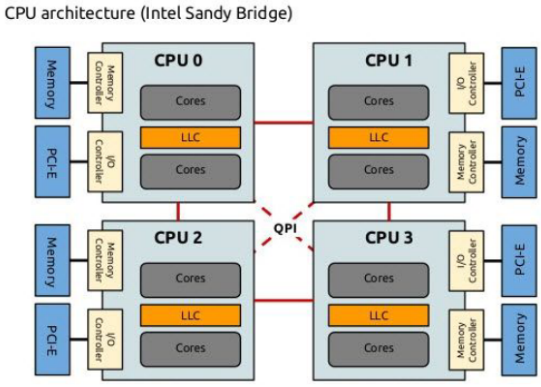


Figure 2: Non uniform memory access Design

Challenges in Multiprogramming

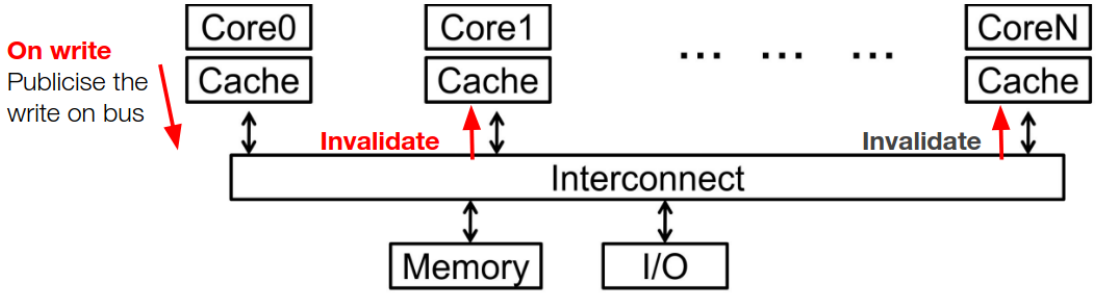
Multiprogramming is challenging due to its non-deterministic behavior, leading to bugs that are hard to reproduce and fix. System behavior can vary, making it difficult to verify the correctness of programs.

Making Multiprogramming Predictable: - *Locks:* Ensuring thread synchronization by maintaining order, even in parallel execution.

- **Problems with Locks:** - Threads waiting for locks may sleep, causing context-switching overheads, adding significant latency (up to 20,000 instructions).
- **Mutexes vs. Spinlocks**: Choosing between CPU overhead due to spinlocks or context-switch overhead with mutexes.

Cache Coherence

Consider an L1 cache which is typically not shared across the cores. Now if some core modifies an entry in its cache and some other core which has the same entry in its L1 cache accesses it, there will be inconsistencies. Cache coherenece solves the problem. Cache coherence ensures that the value at a memory location remains consistent across all cores. If one core modifies a memory location, it must propagate this change to other cores to avoid inconsistencies.



- When a write occurs, the cache broadcasts the change on the bus, invalidating other caches holding the stale value, thus making things consistent.

So the implementation of a lock below has some problem:

```
int value = 0;
acquire () {
    while (test&set(value));
}
release () {
 value = 0;
}
```

When the test&set function is called, it always does a write operation leading to frequent changes. Due to this, the bus above might get flooded with invalid messages, called an **invalidation storm**

Better Lock Implementation

```
int lock = 0; // Free
acquire() {
    do {
        while(lock); // Wait until might be free
    } while(test&set(&lock)); // exit if get lock
}
release() {
    lock = 0;
}
```

This lock implementation reduces bus traffic by minimizing writes, avoiding the invalidation storm.

Remaining Issues: - *False Sharing:* Modern processors have cache lines larger than a word size. Invalidation of entire cache lines, even for a single word change, causes unnecessary overhead. The solution is to carefully align data structures to avoid false sharing.

Per-Core Sharding

Per-core sharding schedules one thread per core, using message passing for communication instead of shared memory. This eliminates the need for locking and reduces cache coherence overhead.

Drawbacks: - I/O and other factors can become bottlenecks in a purely sharded system. The implementation is no longer simple and requires async programming (Ex: ScyllaDB uses per-core sharding using seastar framework).

Seastar Framework

Seastar is a C++ library for asynchronous programming, similar to Python's asyncio, designed to aid multicore programming. It uses **futures** and **promises** to manage asynchronous tasks, which helps in reducing the overhead associated with context switching in traditional multithreading.

Seastar runs on a sharded architechture which avoids synchronization needs like locks.

Setup Instructions:

It took quite a while for me to setup seastar, so I have penned down an easy setup method.
1. Install seastar from their official website/ clone from github repo. https://docs.seastar.io
2. Update your Project's CMakeLists.txt as follows:

```
cmake_minimum_required(VERSION 3.5)
project(SeastarExample)
find_package(Seastar REQUIRED)
add_executable(example test.cpp)
target_link_libraries(example PRIVATE Seastar::seastar)
```

Replace the test.cpp with the file which you want to compile.
3. Modify `/proc/sys/fs/aio-max-nr` to `65536*2` for better performance.
4. Build and run the program:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ ./example
```

Hello World Example in Seastar:

```
#include "seastar/core/app-template.hh"
#include "seastar/core/reactor.hh"
#include <iostream>

int main(int argc, char ** argv){
   seastar::app_template app;
   app.run(argc, argv, []{
       std::cout << "Hello World!" << std::endl;
       return seastar::make_ready_future<>();
   });
}
```

An interesting example

```
#include <seastar/core/app-template.hh>
#include <seastar/core/reactor.hh>
#include <iostream>

int main(int argc, char** argv) {
    seastar::app_template app;
    app.run(argc, argv, [] {
            std::cout << seastar::smp::count << "\n";
            auto cpu_list = seastar::smp::all_cpus();
```

```
            for(auto &it:cpu_list){
                std::cout<<it<<" ";
            }
            std::cout<<std::endl;
            return seastar::make_ready_future<>();
    });
}
```

The first line will print the number of cores detected by seastar
The second line will return a vector containing the cores

```
shiva@shiva-OMEN-by-HP-Gaming-Laptop-16-wd0xxx:~/Desktop/docs/build$ ./example
WARN  2024-09-02 20:51:41,114 seastar - Your system does not have enough AIO capa
WARN  2024-09-02 20:51:41,114 seastar - Resultant AIO control block usage:
WARN  2024-09-02 20:51:41,114 seastar -
WARN  2024-09-02 20:51:41,114 seastar - purpose  per cpu  all 12 cpus
WARN  2024-09-02 20:51:41,114 seastar - -------  -------  -----------
WARN  2024-09-02 20:51:41,114 seastar - storage     1024        12288
WARN  2024-09-02 20:51:41,114 seastar - preempt        2           24
WARN  2024-09-02 20:51:41,114 seastar - network     4435        53220
WARN  2024-09-02 20:51:41,114 seastar - -------  -------  -----------
WARN  2024-09-02 20:51:41,114 seastar - total       5461        65532
WARN  2024-09-02 20:51:41,114 seastar -
WARN  2024-09-02 20:51:41,114 seastar - For optimal network performance, set /pro
INFO  2024-09-02 20:51:41,114 seastar - Reactor backend: io_uring
INFO  2024-09-02 20:51:41,118 seastar - Perf-based stall detector creation failed
to posix timer.
12
0 1 2 3 4 5 6 7 8 9 10 11
shiva@shiva-OMEN-by-HP-Gaming-Laptop-16-wd0xxx:~/Desktop/docs/build$ ./example
INFO  2024-09-02 20:52:09,905 seastar - Reactor backend: io_uring
INFO  2024-09-02 20:52:09,908 seastar - Perf-based stall detector creation failed
to posix timer.
12
0 1 2 3 4 5 6 7 8 9 10 11
```

Figure 3: Results on my machine

For more details, refer to the https://docs.seastar.io/master/tutorial.html The docs are easy to follow.