Managing Shared State

Godmar Back

Virginia Tech

October 8, 2024

Introduction

- Because multi-threaded programs share a virtual address space, they can directly access all objects allocated in the address space: global variables, heap objects, etc.
- The non-determinism introduced by a preemptive scheduling regime and/or the simultaneous execution on multiple processors or cores makes such sharing challenging
- Motivating example: `counter++`

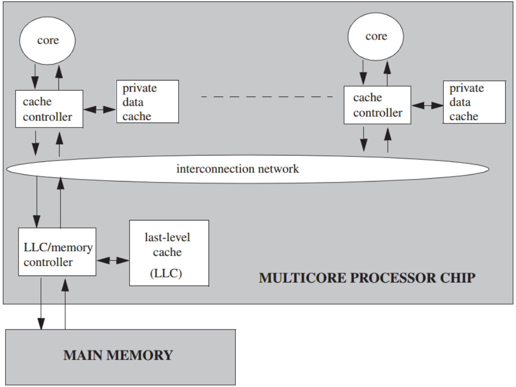


Figure 1: Typical Shared Memory Multicore Architecture, Sorin 2011 [2]

Data Races

Definition

Two or more threads access a shared variable, at least one access is a write access, and final result depends on the order of the execution of those threads, and specifically the order of their memory accesses

- Such data races are one form of concurrency-related bugs (aka "race conditions")
- Can be intermittent and difficult to find ("Heisenbugs")
- Others include: ordering violations, atomicity violations
- Will discuss the following strategies:
  - Avoidance by duplication & partitioning
  - Use of mutual exclusion devices, e.g., locks
  - Use of atomic instructions

Avoiding Sharing

- Data races cannot occur during those parts of a computation where no data is shared
- Strategies to reformulate the problem include:
  - Partitioning: separate data into disjoint parts operated on separately by multiple threads

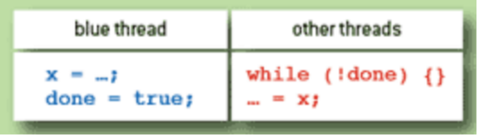
¼	¼	¼	¼

  - Duplicating: provide separate copies of a data structure that threads can independently modify
- Many scalability breakthroughs were obtained from redesigning data structures to avoid or reduce sharing, e.g.
  - Per-thread counters
  - Per-CPU ready queues
  - Region-based memory allocators
- However, these strategies can incur a cost in terms of management complexity and require additional space

Ad-hoc Strategies

There are, unfortunately, many myths about what is safe and what isn't when it comes to writing data race free programs.

- Myth 1: "I'm just reading, not updating"
- Myth 2: "By cleverly arranging the order of read/write accesses I can avoid a data race"
- Myth 3: "I can avoid races with `volatile`"



blue thread	other threads

`x = …;`
`done = true;`
	`while (!done) {}`
	`… = x;`

Figure 2: Is there a data race on x? Source: [1]

Ad-hoc Strategies Fail

waitingonaflag.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool done;
int x;

void thread1() {
    x = rand() % 2;
    done = true;
}

int thread2() {
    while (!done) { }
    return x;
}
```

compiled with gcc 7.5

```
thread1:
    subq    $8, %rsp
    call    rand@PLT
    movl    %eax, %edx
    movb    $1, done(%rip)   # done = true
    shrl    $31, %edx
    addl    %edx, %eax
    andl    $1, %eax
    subl    %edx, %eax
    movl    %eax, x(%rip)    # x = ...
    addq    $8, %rsp
    ret

thread2:
    cmpb    $0, done(%rip)
    jne .L8         # if !done goto L8
.L7:                # else: loop forever
    jmp .L7
.L8:
    movl    x(%rip), %eax
    ret
```

- compiler reorders statements
- compiler replaces loops with ifs

Sequential Consistency

- Consider each thread's program a series of "steps"
- A sequentially consistent execution is the result of some interleaving of these steps (without changing the order within each thread, and immediately propagating updates to other threads)
- By default, programming languages do not guarantee such an execution because
  - Compiler may reorder statements that do not logically depend on each other
  - Processors may reorder load/store instructions likewise
- See Adve & Boehm [1] for precise definition and discussion

Rule

Modern languages such as C, C++, Java guarantee sequentially consistent execution only for programs that are data race free. Achieving freedom from data races requires synchronization, specifically the proper use of critical sections (aka mutual exclusion, locks)

Mutual Exclusion

- Traditionally known as the "critical section" problem
- Idea:
  - Threads update shared data only after entering a critical section
  - Only 1 thread can be inside a critical section at a time
  - Thread enter critical sections only when accessing shared data
- Thus, all operations (reads and writes) performed within a critical section appear as one atomic (isolated) unit
- Provided by mutexes, commonly called "locks"
  - An *entry* operation "acquires," or locks the lock
  - The thread is then said to "hold" the lock
  - An *exit* operation "releases," or unlocks the lock

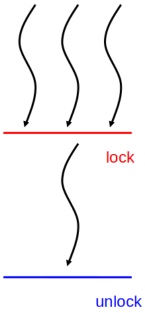


Figure 3: Only 1 thread may acquire a lock at a time

Understanding Locks

- Locks do not protect code. They protect data.
- Each piece of shared data needs to have a lock that protects it:

```
struct list task_queue; // protected by task_queue_lock
pthread_mutex_t task_queue_lock; // protects task_queue
```

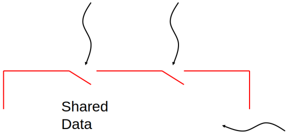
- Locks do not protect data automatically – rather, they enable a cooperative protocol between threads to avoid data races on data they share
- **No matter where in the code the data is accessed** *that associated* lock must be held
- Different pieces of data may be protected by the same lock[1]

Ignoring these rules accounts for most intermittent failures that involve data corruption in our student projects

[1] In fact, break up locks only when necessary

Understanding Locks: Dont's

- Do not use different locks in different sections of your code when accessing the same data



- Do not try to acquire a lock you already hold[2]
- Do not forget to unlock a lock on some path
  - Best practice is to ensure the invariant at the function level: either a lock is not held upon entry and not held upon exit; or it is assumed to be held upon entry and stays locked upon exit.
- Do not hold a lock while blocking for some event (sleeping, I/O, ...)
  - This would make threads that need the same lock wait for the same event

[2] unless recursive mutexes are used

Understanding Locks: Do's

- Use race condition detection tools, e.g. Helgrind, DRD, Intel Thread Checker, TSan

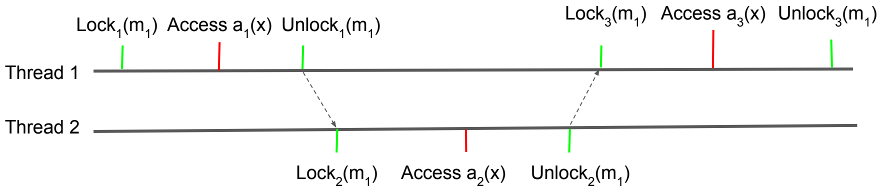


Figure 4: When using proper locking discipline, a pairwise "happens-before" relationship is established between all pairs of accesses to variable x; e.g.: $a_1(x) \rightarrow U_1(m_1)$ because it's done by the same thread (ditto for $L_2(m_1) \rightarrow a_2(x) \rightarrow U_2(m_1)$ and $L_3(m_1) \rightarrow a_3(x)$). The relationships $U_1(m_1) \rightarrow L_2(m_2)$ and $U_2(m_1) \rightarrow L_3(m_1)$ hold because unlocking a mutex happens before the next successful lock operation. $a_k(x) \parallel a_j(x)$ iff neither $a_k(x) \rightarrow a_j(x)$ nor $a_j(x) \rightarrow a_k(x)$.

Thread-Safety

Definition

Property of a function to yield correct result when called from multiple threads

- Attribute that must be documented as part of the API documentation of a library
  - Some traditional C functions are not thread-safe (`strtok`)
- Functions are not thread-safe if they
  1. Fail to protect shared variables
  2. Rely on persistent state across invocations
  3. Return a pointer to a static variable
  4. Call other functions that aren't thread safe

Relying on persistent state across invocations

- Not only does this function fail to protect shared state (next), it also does not maintain deterministic order in which pseudo-random numbers are being generated.
- Fix: pass state to function, e.g. rand_r()
- With the introduction of support for multithreading, the C library was updated with several _r() functions that are thread-safe
- The 'r' stands for reentrant, which is a stronger property than being thread-safe. Reentrant functions can be safely reentered while a call is in progress - as, for example, in the case of recursive functions.

Example: rand()

```
static unsigned int next = 1;

/* rand -return pseudo-random integer
   in 0..32767 */
int rand(void) {
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand-set seed for rand() */
void srand(unsigned intseed) {
    next = seed;
}
```

Fix: rand_r()

```
int rand_r(unsigned int *seedp);
```

References

[1] Hans-J. Boehm and Sarita V. Adve.
You don't know jack about shared variables or memory models.
Commun. ACM, 55(2):48–54, February 2012.

[2] Daniel J. Sorin, Mark D. Hill, and David A. Wood.
A Primer on Memory Consistency and Cache Coherence.
Morgan & Claypool Publishers, 1st edition, 2011.