Design of Parallel and High-Performance Computing

Fall 2013

Lecture: Languages and Locks

Instructor: Torsten Hoefler & Markus Püschel

TA: Timo Schneider

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Administrivia

- **You should have a project partner by now**
  - Make sure, Timo knows about your team (this step is **important**!)
  - Think about a project
- **Initial project presentations: Monday 11/4 during lecture**
  - Send slides (ppt or pdf) by 11/3 11:59pm to Timo!
  - 10 minutes per team (hard limit)
  - **Prepare**! This is your first impression, gather feedback from us!
  - Rough guidelines:
    Present your plan
    Related work (what exists, literature review!)
    Preliminary results (not necessarily)
    Main goal is to gather feedback, so present some details
    Pick one presenter (make sure to switch for other presentations!)
- **Intermediate (very short) presentation: Thursday 11/21 during recitation**
- **Final project presentation: Monday 12/16 during last lecture**

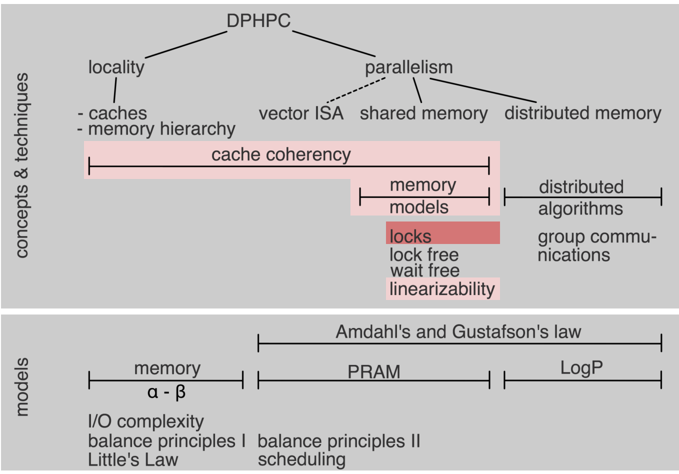
Distinguished Colloquium

- **Right after our lecture in CAB G61**
- **Luis Ceze: Disciplined Approximate Computing: From Language to Hardware, and Beyond**
- **Will add one more parameter to computing: reliability**
  - Very interesting, you should all go!

Review of last lecture

- **Locked Queue**
  - Correctness
  - Lock-free two-thread queue
- **Linearizability**
  - Combine object pre- and postconditions with serializability
  - Additional (semantic) constraints!
- **Histories**
  - Analyze given histories
    Projections, Sequential/Concurrent, Completeness, Equivalence, Well formed, Linearizability (formal)
- **Language memory models**
  - History
  - Java/C++ overview

DPHPC Overview



Goals of this lecture

- **Languages and Memory Models**
  - Java/C++ definition
- **Recap serial consistency**
  - Races (now in practice)
- **Mutual exclusion**
- **Locks**
  - Two-thread
  - Peterson
  - N-thread
  - Many different locks, strengths and weaknesses
  - Lock options and parameters
- **Problems and outline to next class**

Java and C++ High-level overview

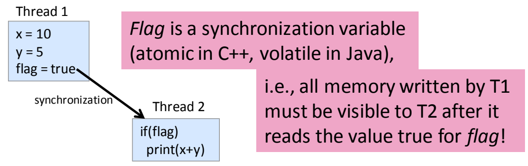
- **Relaxed memory model**
  - No global visibility ordering of operations
  - Allows for standard compiler optimizations
- **But**
  - Program order for each thread (sequential semantics)
  - Partial order on memory operations (with respect to synchronizations)
  - Visibility function defined
- **Correctly synchronized programs**
  - Guarantee sequential consistency
- **Incorrectly synchronized programs**
  - Java: maintain safety and security guarantees

    Type safety etc. (require behavior bounded by causality)
  - C++: undefined behavior

    No safety (anything can happen/change)

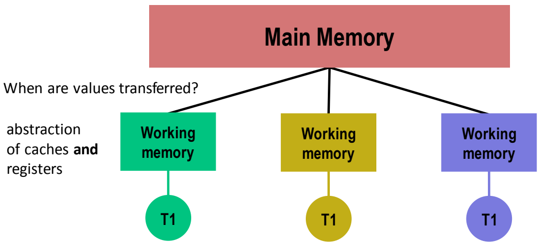
Communication between Threads: Intuition

- **Not guaranteed unless by:**
  - Synchronization
  - Volatile/atomic variables
  - Specialized functions/classes (e.g., java.util.concurrent, …)



Memory Model: Intuition

- **Abstract relation between threads and memory**
  - Local thread view!



- **Does not talk about classes, objects, methods, …**
  - Linearizability is a higher-level concept!

Lock Synchronization

- **Java**

```
synchronized (lock) {
  // critical region
}
```

  - Synchronized methods as syntactic sugar

- **C++**

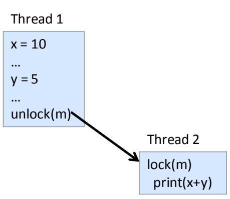
```
{
  unique_lock<mutex> l(lock);
  // critical region
}
```

  - Many flexible variants

- **Semantics:**
  - mutual exclusion
  - at most one thread may own a lock
  - a thread B trying to acquire a lock held by thread A blocks until thread A releases lock
  - note: threads may wait forever (no progress guarantee!)

Memory semantics

- **Similar to synchronization variables**



  - All memory accesses before an unlock …
  - are ordered before and are visible to …
  - any memory access after a matching lock!

Synchronization Variables

- **Variables can be declared volatile (Java) or atomic (C++)**
- **Reads and writes to synchronization variables**
  - Are totally ordered with respect to all threads
  - Must not be reordered with normal reads and writes
- **Compiler**
  - Must not allocate synchronization variables in registers
  - Must not swap variables with synchronization variables
  - May need to issue memory fences/barriers
  - …

Synchronization Variables

- **Write to a synchronization variable**
  - Similar memory semantics as unlock (no process synchronization!)
- **Read from a synchronization variable**
  - Similar memory semantics as lock (no process synchronization!)

```
class example {
  int x = 0;
  atomic<bool> v = false

  public void writer() {
    x = 42;
    v = true;
  }

  public void reader() {
    if(v) {
      print(x)
    }
  }
```

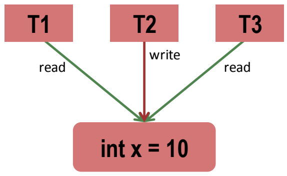
Thread 1

Thread 2

Without volatile, a platform may reorder these accesses!

Memory Model Rules

- **Java/C++: Correctly synchronized programs will execute sequentially consistent**
- **Correctly synchronized = data-race free**
  - iff all sequentially consistent executions are free of data races
- **Two accesses to a shared memory location form a data race in the execution of a program if**
  - The two accesses are from different threads
  - At least one access is a write and
  - The accesses are not synchronized



Locks - Lecture Goals

- **You understand locks in detail**
  - Requirements / guarantees
  - Correctness / validation
  - Performance / scalability
- **Acquire the ability to design your own locks**
  - Understand techniques and weaknesses/traps
  - Extend to other concurrent algorithms

    Issues are very much the same
- **Feel the complexity of shared memory!**

Preliminary Comments

- **All code examples are in C/C++ style**
  - Neither C nor C++ <11 have a clear memory model
  - C++ is one of the languages of choice in HPC
  - Consider source as exemplary (and pay attention to the memory model)!

    In fact, many/most of the examples are incorrect in anything but sequential consistency!

    In fact, you'll never need those algorithms, but the principles demonstrated!
- **x86 is really only used because it's common**
  - This does not mean that we consider the ISA or memory model elegant!
  - We assume atomic memory (or registers)!

    Usually given on x86 (easy to enforce)
- **Number of threads/processes is p, tid is the thread id**

Recap Concurrent Updates

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

gcc -O3

```
    movl  $1000, %eax     // i=n=1000
.L2:
    movl  (%rdx), %ecx    // ecx = *a
    addl  $1, %ecx        // ecx++
    subl  $1, %eax        // i—
    movl  %ecx, (%rdx)    // *a = ecx
    jne   .L2             // loop if i>0
                          [sub sets ZF]
```

- **Multi-threaded execution!**
  - Value of a for p=1?
  - Value of a for p>1?

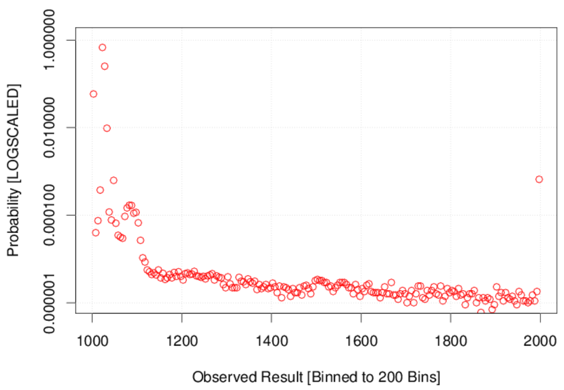
    Why? Isn't it a single instruction?

Some Statistics

- **Nondeterministic execution**
  - Result depends on timing (probably not desired)
- **What do you think are the most significant results?**
  - Running two threads on Core i5 dual core
  - a=1000? 2000? 1500? 1223? 1999?

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

Some Statistics

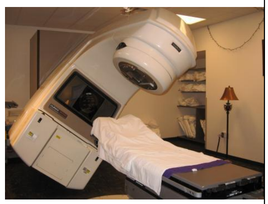


Conflicting Accesses

- **(recap) two memory accesses conflict if they can happen at *the same time* (in happens-before) and one of them is a write (store)**
- **Such a code is said to have a "race condition"**
  - Also data-race
  - Trivia around races:

    The Therac-25 killed three people due to a race

    A data-race lead to a large blackout in 2003, leaving 55 million people without power causing $1bn damage
- **Can be avoided by critical regions**
  - Mutually exclusive access to a set of operations



Mutual Exclusion

- **Control access to a critical region**
  - Memory accesses of all processes happen in program order (a partial order, many interleavings)

    An execution defines a total order of memory accesses
  - Some subsets of memory accesses (issued by the same process) need to happen atomically (thread a's memory accesses may not be interleaved with other thread's accesses)

    We need to restrict the valid executions
- **→ Requires synchronization of some sort**
  - Many possible techniques (e.g., TM, CAS, T&S, …)
  - We discuss locks which have wait semantics

```
    movl    $1000, %eax     // i=1000
.L2:
    movl  (%rdx), %ecx    // ecx = *a
    addl  $1, %ecx        // ecx++
    subl  $1, %eax        // i—
    movl  %ecx, (%rdx)    // *a = ecx
    jne   .L2             // loop if i>0
                          [sub sets ZF]
```


Fixing it with locks

```
const int n=1000;
volatile int a=0;
omp_lock_t lck;
for (int i=0; i<n; ++i) {
  omp_set_lock(&lck);
  a++;
  omp_unset_lock(&lck);
}
```

gcc -O3

```
    movl  $1000, %ebx    // i=1000
.L2:
    movq  0(%rbp), %rdi   // (SystemV CC)
    call  omp_set_lock   // get lock
    movq  0(%rbp), %rdi  // (SystemV CC)
    movl  (%rax), %edx   // edx = *a
    addl  $1, %edx       // edx++
    movl  %edx, (%rax)   // *a = edx
    call  omp_unset_lock // release lock
    subl  $1, %ebx       // i—
    jne   .L2            // repeat if i>0
```

- **What must the functions lock and unlock guarantee?**
  - **#1: prevent two threads from simultaneously entering CR**

    i.e., accesses to CR must be mutually exclusive!
  - **#2: ensure consistent memory**

    i.e., stores must be globally visible before new lock is granted!

Lock Overview

- **Lock/unlock or acquire/release**
  - Lock/acquire: before entering CR
  - Unlock/release: after leaving CR
- **Semantics:**
  - Lock/unlock pairs have to match
  - Between lock/unlock, a thread holds the lock

Lock Properties

- **Mutual exclusion**
  - Only one thread is on the critical region
- **Consistency**
  - Memory operations are visible when critical region is left
- **Progress**
  - If any thread a is not in the critical region, it cannot prevent another thread b from entering
- **Starvation-freedom (implies deadlock-freedom)**
  - If a thread is requesting access to a critical region, then it will eventually be granted access
- **Fairness**
  - A thread a requested access to a critical region before thread b. Did is also granted access to this region before b?
- **Performance**
  - Scaling to large numbers of contending threads

?

Notation

- **Time defined by precedence (a total order on events)**
  - Events are instantaneous
  - Threads produce sequences of events $a_0, a_1, a_2, \ldots$
  - Program statements may be repeated, denote i-th instance of a as a^i
  - Event a occurs before event b: a → b
  - An interval (a,b) is the duration between events a → b
  - Interval I_1=(a,b) precedes interval I_2=(c,d) iff b → c
- **Critical regions**
  - A critical region CR is an interval a → b, where a is the first operation in the CR and b the last
- **Mutual exclusion**
  - Critical regions CR_A and CR_B are mutually exclusive if:

    Either CR_A → CR_B or CR_B → CR_A for all instances!
- **Assume atomic registers (for now)**

Simple Two-Thread Locks

- **A first simple spinlock**

```
volatile int flag=0;

void lock(lock) {
  while(flag);
  flag = 1;
}

void unlock (lock) {
  flag = 0;
}
```

Busy-wait to acquire lock (spinning)

Is this lock correct?

Why does this not guarantee mutual exclusion?

Proof Intuition

- **Construct a sequentially consistent order that permits both processes to enter CR**

Simple Two-Thread Locks

- **Another two-thread spin-lock: LockOne**

```
volatile int flag[2];

void lock() {
  int j = 1 - tid;
  flag[tid] = true;
  while (flag[j]) {} // wait
}

void unlock() {
  flag[tid] = false;
}
```

When and why does this guarantee mutual exclusion?

Correctness Proof

- **In sequential consistency!**
- **Intuitions:**
  - Situation: both threads are ready to enter
  - Show that situation that allows both to enter leads to a schedule violating sequential consistency

    Using transitivity of happens-before relation

Simple Two-Thread Locks

- **Another two-thread spin-lock: LockOne**

```
volatile int flag[2];

void lock() {
  int j = 1 - tid;
  flag[tid] = true;
  while (flag[j]) {} // wait
}

void unlock() {
  flag[tid] = false;
}
```

When and why does this guarantee mutual exclusion?

Does it work in practice?

Simple Two-Thread Locks

- **A third attempt at two-thread locking: LockTwo**

```
volatile int victim;

void lock() {
  victim = tid; // grant access
  while (victim == tid) {} // wait
}

void unlock() {}
```

Does this guarantee mutual exclusion?

Correctness Proof

- **Intuition:**
  - Victim is only written once per lock()
  - A can only enter after B wrote
  - B cannot enter in any sequentially consistent schedule

Simple Two-Thread Locks

- **A third attempt at two-thread locking: LockTwo**

```
volatile int victim;

void lock() {
  victim = tid; // grant access
  while (victim == tid) {} // wait
}

void unlock() {}
```

Does this guarantee mutual exclusion?

Does it work in practice?

Simple Two-Thread Locks

- **The last two locks provide mutual exclusion**
  - LockOne succeeds iff lock attempts overlap
  - LockTwo succeeds iff lock attempts do not overlap
- **Combine both into one locking strategy!**
  - Peterson's lock (1981)

Peterson's Two-Thread Lock (1981)

- **Combines the first lock (request access) with the second lock (grant access)**

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I'm interested
  victim = tid;      // other goes first
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0;  // I'm not interested
}
```


Proof Correctness

- **Intuition:**
  - Victim is written once
  - Pick thread that wrote victim last
  - Show thread must have read flag==0
  - Show that no sequentially consistent schedule permits that

Starvation Freedom

- **(recap) definition: Every thread that calls lock() eventually gets the lock.**
  - Implies deadlock-freedom!
- **Is Peterson's lock starvation-free?**

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I'm interested
  victim = tid;      // other goes first
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0;  // I'm not interested
}
```


Proof Starvation Freedom

- **Intuition:**
  - Threads can only wait/starve in while()
    Until flag==0 or victim==other
  - Other thread enters lock() → sets victim to other
    Will definitely "unstuck" first thread
  - So other thread can only be stuck in lock()
    Will wait for victim==other, victim cannot block both threads → one must leave!

Peterson in Practice … on x86

- **Implement and run on x86**
- **100000 iterations**
  - 1.6 · 10^{-6}% errors
  - What is the problem?

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I'm interested
  victim = tid;      // other goes first
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0;  // I'm not interested
}
```


Peterson in Practice … on x86

- **Implement and run on x86**
- **100000 iterations**
  - 1.6 · 10^{-6}% errors
  - What is the problem?
    No sequential consistency for W(flag[tid]) and R(flag[j])

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I'm interested
  victim = tid;      // other goes first
  asm ("mfence");
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0;  // I'm not interested
}
```


Peterson in Practice … on x86

- **Implement and run on x86**
- **100000 iterations**
  - 1.6 · 10^{-6}% errors
  - What is the problem?
    No sequential consistency for W(flag[tid]) and R(flag[j])
  - Still 1.3 · 10^{-6}%
    Why?

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I'm interested
  victim = tid;      // other goes first
  asm ("mfence");
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0;  // I'm not interested
}
```


Peterson in Practice … on x86

- **Implement and run on x86**
- **100000 iterations**
  - 1.6 · 10^{-6}% errors
  - What is the problem?
    No sequential consistency for W(flag[tid]) and R(flag[j])
  - Still 1.3 · 10^{-6}%
    Why?
    Reads may slip into CR!

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I'm interested
  victim = tid;      // other goes first
  asm ("mfence");
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  asm ("mfence");
  flag[tid] = 0;  // I'm not interested
}
```

Correct Peterson Lock on x86

- **Unoptimized (naïve sprinkling of mfences)**
- **Performance:**
  - No mfence
    375ns
  - mfence in lock
    379ns
  - mfence in unlock
    404ns
  - Two mfence
    427ns (+14%)

```
volatile int flag[2];
volatile int victim;

void lock() {
 int j = 1 - tid;
 flag[tid] = 1; // I'm interested
 victim = tid;     // other goes first
 asm ("mfence");
 while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
 asm ("mfence");
 flag[tid] = 0;  // I'm not interested
}
```


Locking for N threads

- **Simple generalization of Peterson's lock, assume n levels l = 0…n-1**
  - Is it correct?

```
volatile int level[n] = {0,0,…,0}; // indicates highest level a thread tries to enter
volatile int victim[n]; // the victim thread, excluded from next level
void lock() {
 for (int i = 1; i < n; i++) { //attempt level i
  level[tid] = i;
  victim[i] = tid;
  // spin while conflicts exist
  while ((∃k != tid) (level[k] >= i && victim[i] == tid )) {};
 }
}

void unlock() {
 level[tid] = 0;
}
```


Filter Lock - Correctness

- **Lemma: For 0<j<n-1, there are at most n-j threads at level j!**
- **Intuition:**
  - Recursive proof (induction on j)
  - By contradiction, assume n-j+1 threads at level j-1 and j
  - Assume last thread to write victim
  - Any other thread writes level before victim
  - Last thread will stop at spin due to other thread's write
- **j=n-1 is critical region**

Locking for N threads

- **Simple generalization of Peterson's lock, assume n levels l = 0…n-1**
  - Is it starvation-free?

```
volatile int level[n] = {0,0,…,0}; // indicates highest level a thread tries to enter
volatile int victim[n]; // the victim thread, excluded from next level
void lock() {
 for (int i = 1; i < n; i++) { //attempt level i
  level[tid] = i;
  victim[i] = tid;
  // spin while conflicts exist
  while ((∃k != tid) (level[k] >= i && victim[i] == tid )) {};
 }
}

void unlock() {
 level[tid] = 0;
}
```


Filter Lock Starvation Freedom

- **Intuition:**
  - Inductive argument over j (levels)
  - Base-case: level n-1 has one thread (not stuck)
  - Level j: assume thread is stuck
    Eventually, higher levels will drain (induction)
    One thread x sets level[x] to j
      Eventually, no more threads enter level j
    Victim can only have one value → one thread will advance!

Filter Lock

- **What are the disadvantages of this lock?**

```
volatile int level[n] = {0,0,…,0}; // indicates highest level a thread tries to enter
volatile int victim[n]; // the victim thread, excluded from next level
void lock() {
 for (int i = 1; i < n; i++) { // attempt level i
  level[tid] = i;
  victim[i] = tid;
  // spin while conflicts exist
  while ((∃k != tid) (level[k] >= i && victim[i] == tid )) {};
 }
}

void unlock() {
 level[tid] = 0;
}
```

Lock Fairness

- **Starvation freedom provides no guarantee on how long a thread waits or if it is "passed"!**
- **To reason about fairness, we define two sections of each lock algorithm:**
  - Doorway D (bounded # of steps)
  - Waiting W (unbounded # of steps)

```
void lock() {
  int j = 1 - tid;
  flag[tid] = true; // I'm interested
  victim = tid;     // other goes first
  while (flag[j] && victim == tid) {};
}
```

- **FIFO locks:**
  - If T_A finishes its doorway before T_B the $CR_A \rightarrow CR_B$
  - Implies fairness

Lamport's Bakery Algorithm (1974)

- **Is a FIFO lock (and thus fair)**
- **Each thread takes number in doorway and threads enter in the order of their number!**

```
volatile int flag[n] = {0,0,…,0};
volatile int label[n] = {0,0,….,0};

void lock() {
  flag[tid] = 1; // request
  label[tid] = max(label[0], ...,label[n-1]) + 1; // take ticket
  while ((∃k != tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {};
}
public void unlock() {
  flag[tid] = 0;
}
```


Lamport's Bakery Algorithm

- **Advantages:**
  - Elegant and correct solution
  - Starvation free, even FIFO fairness
- **Not used in practice!**
  - Why?
  - Needs to read/write N memory locations for synchronizing N threads
  - Can we do better?

    Using only atomic registers/memory

A Lower Bound to Memory Complexity

- **Theorem 5.1 in [1]: *"If S is a [atomic] read/write system with at least two processes and S solves mutual exclusion with global progress [deadlock-freedom], then S must have at least as many variables as processes"***
- **So we're doomed! Optimal locks are available and they're fundamentally non-scalable. Or not?**

[1] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. Information and Computation, 107(2):171–184, December 1993

Hardware Support?

- **Hardware atomic operations:**
  - Test&Set

    Write const to memory while returning the old value
  - Atomic swap

    Atomically exchange memory and register
  - Fetch&Op

    Get value and apply operation to memory location
  - Compare&Swap

    Compare two values and swap memory with register if equal
  - Load-linked/Store-Conditional LL/SC

    Loads value from memory, allows operations, commits only if no other updates committed → mini-TM
  - Intel TSX (transactional synchronization extensions)

    Hardware-TM (roll your own atomic operations)