Practical Multi-Threading

Dietmar Kühl
Bloomberg L.P.
dietmar.kuehl@gmail.com

Copyright Notice

© 2008 Bloomberg L.P. Permission is granted to copy, distribute, and display this material, and to make derivative works and commercial use of it. The information in this material is provided "AS IS", without warranty of any kind. Neither Bloomberg nor any employee guarantees the correctness or completeness of such information. Bloomberg, its employees, and its affiliated entities and persons shall not be liable, directly or indirectly, in any way, for any inaccuracies, errors or omissions in such information. Nothing herein should be interpreted as stating the opinions, policies, recommendations, or positions of Bloomberg.

Overview

- threading basics:
  - thread objects: thread start, termination
  - synchronization, mutex, condition variable
- outlook: threading building blocks
- NOT: lock-free programming!

Quick Survey

- who uses STL algorithms?
- ... algorithms like sort(), lower_bound(), etc.?
- ... algorithms like copy(), find(), etc.?
- ... for_each()

Multi-Threading Problems

- dead-lock/live-lock
- race condition
- serial execution == no better performance

Thread Objects

- create a new thread: ctor + function object
- thread identity: the thread's ID
- join with a thread
- detach a thread: explicit or by destructor

Creating Threads I

- ctor of std::thread kicks off a new thread
- joinable until detached or moved
- from a function (note: no extern "C"):
  void work() { ... }
  std::thread(work);

Creating Threads II

- from function passing arguments:
  ```
  void work(int a1, int a2);
  std::thread(work, 1, 2);
  ```
- from a bound function:
  ```
  std::thread(std::bind(work, 1, 2));
  ```
- from a function object:
  ```
  struct work { void operator()() { ... } };
  std::thread(work());
  ```

Getting Rid Of Threads

- no cancellation support
  - use implicit termination at end of work
  - use explicit thread communication
- join() a joinable thread object
- detach for implicit clean-up: detach() or dtor

Complete Example

```
bool flag(true);
void work() {
  while (flag) { std::this_thread::sleep(t); }
}
int main() {
  std::thread worker(work);
  std::cin.ignore();
  flag = false;
  worker.join();
}
```

Race Conditions

```
> valgrind -q --tool=helgrind example1
...
Possible data race during write of size 1 at 0xaddr1
    at 0xaddr2: main (example1.cpp:31)
  Old state: owned exclusively by thread #2
  New state: shared-modified by threads #1, #2
  Reason:    this thread, #1, holds no locks at all
```

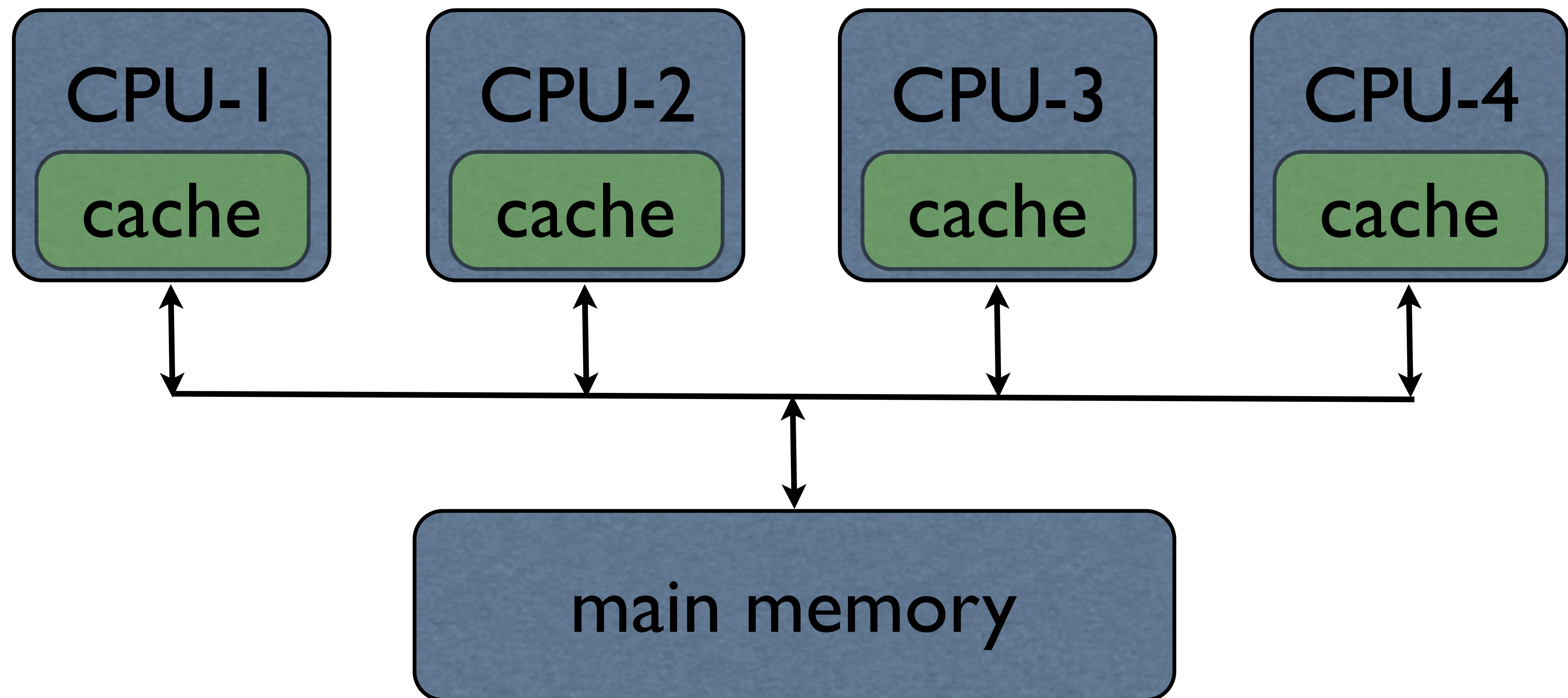
Valgrind/Helgrind

- emulates processors and observes programs behavior
- currently Linux only
- helgrind workable only with 3.3.0 and later
- can do other useful stuff e.g. leak detection

Errors in the Example

```
bool flag(true);
void work() {
  while (flag) { std::this_thread::sleep(t); }
}
int main() {
  std::thread worker(work);
  std::cin.ignore();
  flag = false;
  worker.join();
}
```

Why Is There A Problem?



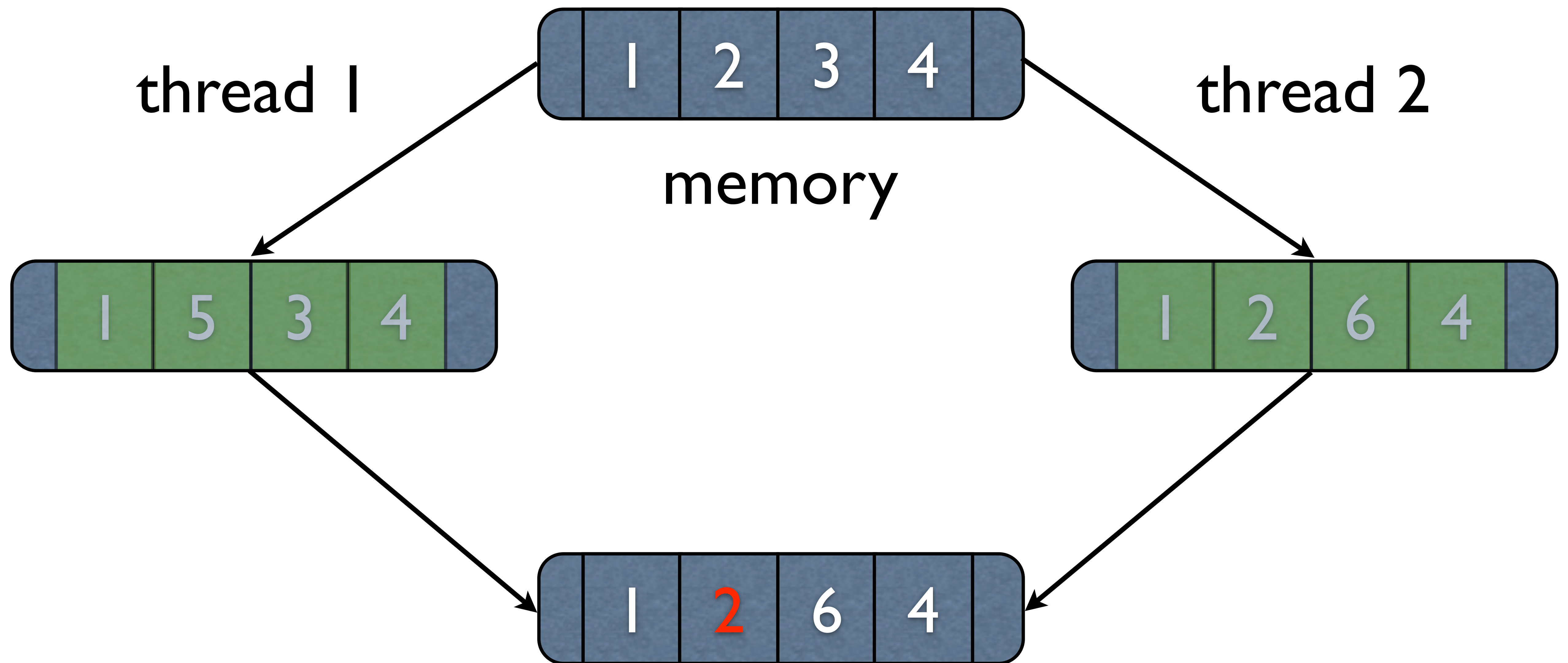
Problems Due To Caching

- reordered writes:

  initial state: bool flag(false); int value(0);
  thread 1: value = 42; flag = 1;
  thread 2: if (flag) std::cout << value << "\n";
  output: ???

- lost updates

Disappearing Writes

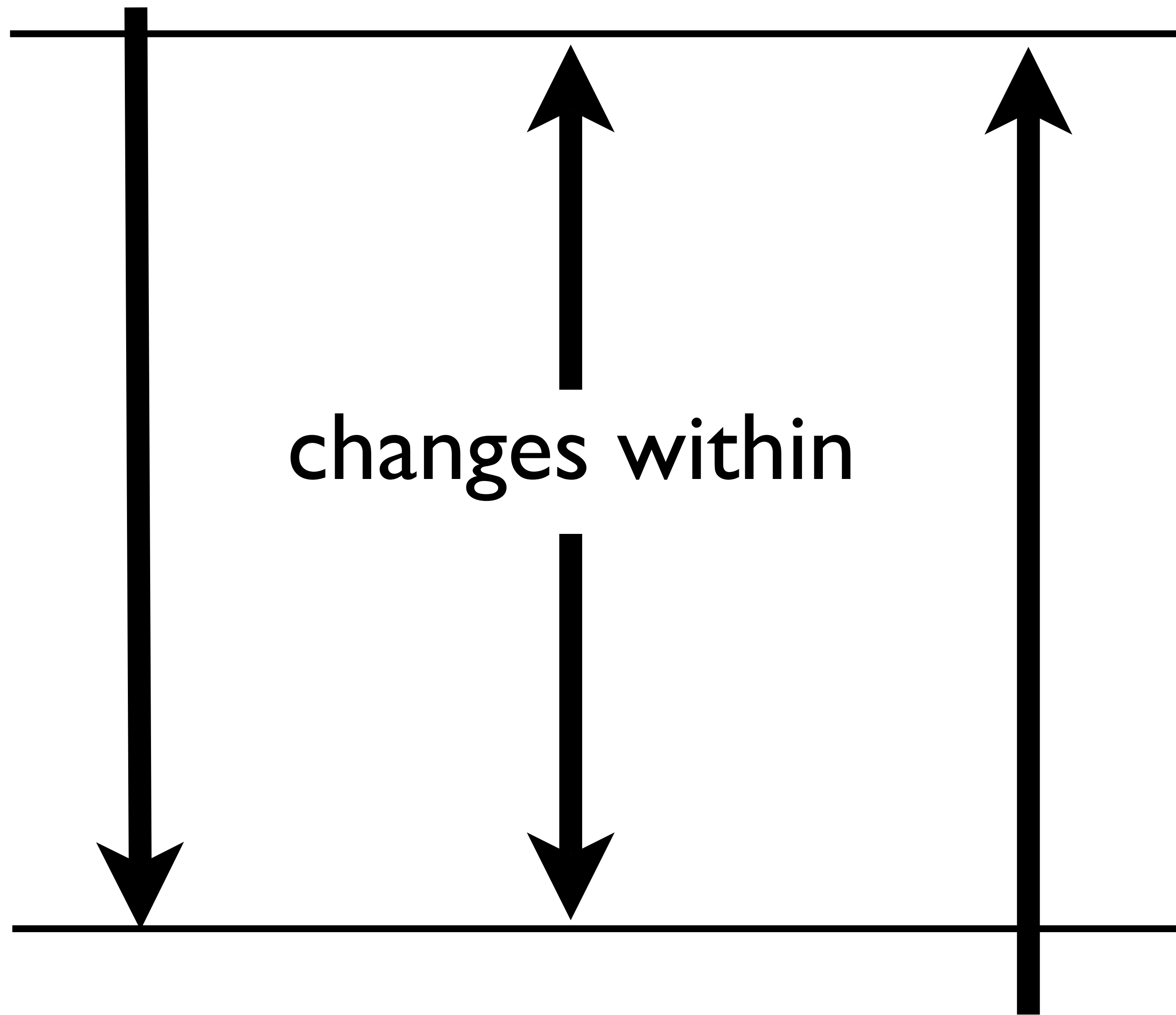


Critical Sections

- any code accessing mutable, shared resources
- always requires some form of synchronization
- has to start with acquisition
- has to end with release
- different models on how this can be done

Critical Section Schema

changes before



changes after

acquire

time

release

Critical Sections: Mutexes

- preferred way for critical sections
- manually using lock()/unlock():
  mutex.lock();
  ...
  mutex.unlock();
- automatic using lock guard:
  std::lock_guard<std::mutex> lock(mutex);
  ...

Critical Sections: Atomics

- limited use in general (best left to experts)
- form critical sections on their own
- critical section (may require memory barriers):

```
std::atomic<bool> flag(false);
while (!flag.load()) {} // acquire;
...                     // critical section
flag.store(false);      // release
```

Synchronization

- sharing mutable data requires synchronization
- ... but not required for immutable data
- synchronization can be provided by
  - ... using mutex, semaphore, etc. locks
  - ... using atomic types or memory barriers

Fixing the Write I

```
std::mutex flag_mutex; bool flag(true);

int main() {
  std::thread worker(work); std::cin.ignore();
  {
    std::lock_guard<std::mutex> lock(flag_mutex);
    flag = false;
  }
  worker.join();
}
```

Fixing the Write II

```
std::mutex flag_mutex; bool flag(true);
void set_flag(bool value) {
  std::lock_guard<std::mutex> lock(flag_mutex);
  flag = value;
}
int main() {
  std::thread worker(work); std::cin.ignore();
  set_flag(false);
  worker.join();
}
```

Fixing the Read

```
std::mutex flag_mutex; bool flag(true);

bool get_flag() {
  std::lock_guard<std::mutex> lock(flag_mutex);
  return flag;
}
void work() {
  while (get_flag()) { std::this_thread::sleep(t); }
}
```

Using Atomic Types

```
std::atomic<bool> flag(true);
void work() { while (flag.load()) { ... } }

int main() {
  std::thread worker(work);
  std::cin.ignore();
  flag.store(false);
  worker.join();
}
```

Mutexes

- THE general purpose synchronization device
- not very expensive but not free either: ~7.5 M/s (uncontended) locks on this machine
- essentially: one mutex for each unit of shared data

Mutex Granularity

- few mutexes => a lot of contention
- each mutex should protect a complete entity:
  - a simple variable: counter, flag
  - communication device: queue
  - a shared data structure

Thread-Safe Interfaces

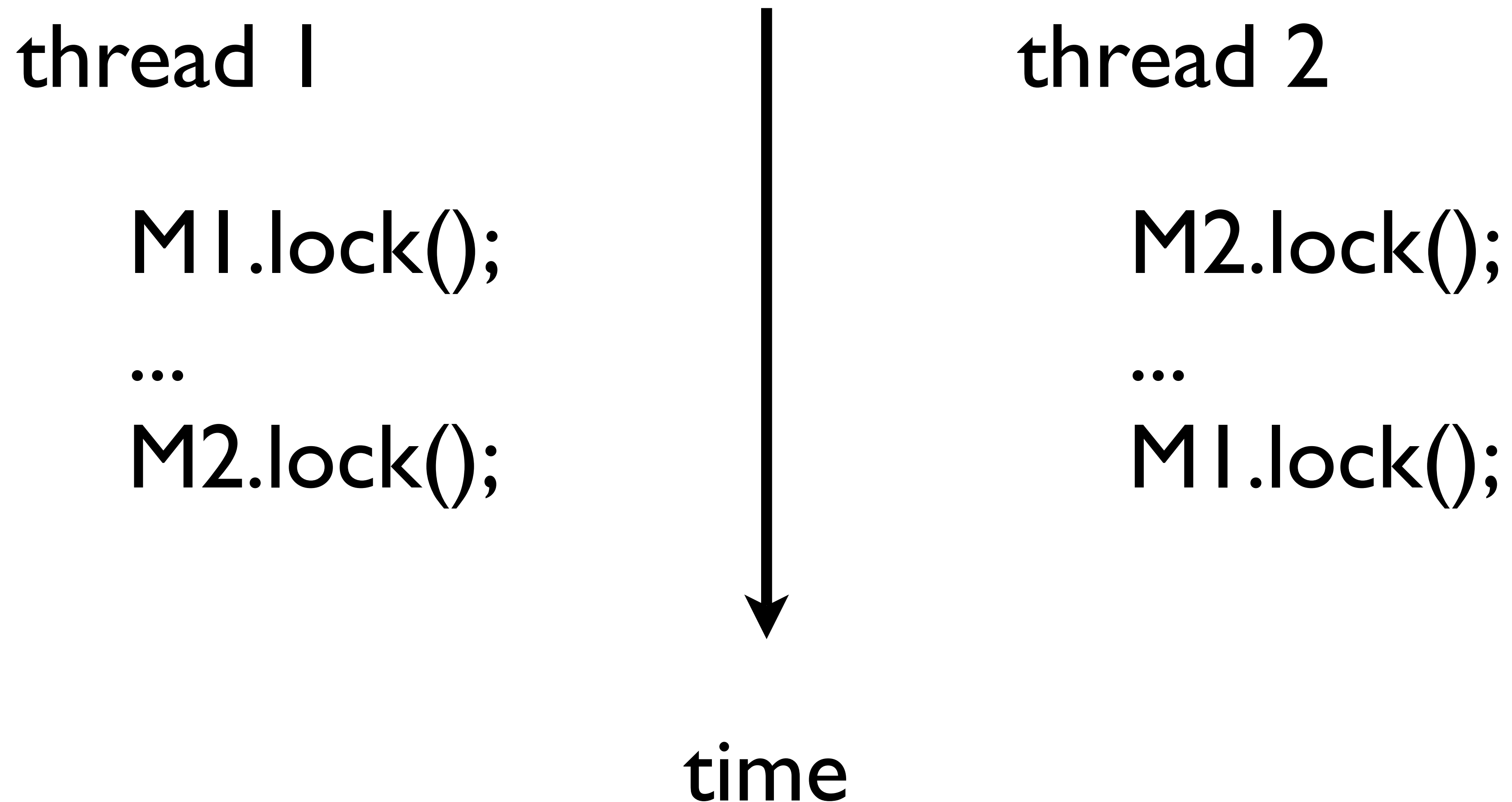
- internal locking (monitor objects):
  - pros: user doesn’t need to know or care
  - cons: useful only for “fire & forget”
  - e.g.: atomics, (non-STL) queue, allocator
- external locking:
  - pros: more flexible
  - cons: more error prone

STL Thread Safety

- user is responsible for proper access
- one container can be ...
  - ... read by multiple threads simultaneously
  - ... written by only one thread and may not be accessed by another thread
- restrictions are per object

Dead-Lock

- threads mutually awaiting release of locks



Mutex Hierarchies

- dead-lock prevention: lock in the same order
- sort mutexes into numbered levels (essentially according to abstraction levels)
- after locking a mutex at level n, only acquire locks for mutexes at lower levels, i.e. < n
- locks at the same level must be acquired

Recursive Mutexes

- obviously not allowed under hierarchy regime
- questionable anyway:
  - critical sections represent transaction
  - within inconsistencies are allowed and likely
  - why use a nested transaction?

Refactoring for Locking

```
void foo() {
  guard l(mutex);
  ...
  bar();
}
void bar() {
  guard l(mutex);
  ...
}
```

```
void foo() {
  guard l(mutex);
  ...
  intern_bar();
}
void bar() {
  guard l(mutex);
  intern_bar();
}
```

Locking and Generic Code

- don’t call unknown code from critical sections
- any such code may lock at the wrong level
- what is generic/unknown code?
  - callback: function pointer, function object
  - virtual function

Try-Lock

- alternate dead-lock prevention: only try-lock
- when locking fails, bail-out of all locks
- ... undoing all work as necessary, of course
- danger: live-lock, i.e two threads starting at the opposite ends continuously failing
- may still cut necessary leeway for calling

Try-Lock Example

```
std::mutex mutex;

std::unique_lock<std::mutex> lock(mutex,
                                  std::try_to_lock);

if (!lock)
  throw std::runtime_error(“already
locked”);

...
```

Condition Variables

- synchronizing actions of threads
  - have multiple threads wait for an event
  - producer/consumer relationship
- event indicator is protected by a mutex
- threads waiting for the event are asleep

Two Roles

- event consumer (while holding a mutex lock)
  - checks condition
  - wait()s if the condition is not met
- event producer
  - [possibly] changes the condition
  - either signals one thread: notify_one()
  - or broadcasts to all threads: notify_all()

The Producer

- may change condition (while holding a lock)
- sends notify, probably not holding the lock

```
std::queue<int> q; std::mutex mutex;
std::condition_variable condition;
void add_message(int message) {
  { std::lock_guard<std::mutex> lock(mutex);
    q.push(message); }
    condition.notify_one();
}
```

The Consumer

- gets lock, checks conditions, waits if not

```
std::queue<int> q; std::mutex mutex;
std::condition_variable condition;
int pop_message() {
  std::unique_lock<std::mutex> lock(mutex);
  while (q.empty()) condition.wait(lock);
  int rc = q.back(); q.pop_back(); return rc;
}
```

Notes on the Consumer

- holding the mutex lock
  - the lock is held when calling wait()!
  - it gets unlocked while waiting
  - it is locked again when wait() returns
- there may be spurious returns from wait()
  - the condition needs to be rechecked!

Spurious Wake-Ups

- producer notifies without meeting condition
- different thread may acquire the lock first
- signal may have been received by the thread
- rule avoids problems with implementations

Summary of Basics

- synchronize shared resources
  - mutexes, atomics, thread start/ termination
- try to avoid use of shared resources
  - stay within cache, don’t risk contention
- synchronize threads: condition variables

Implicit Parallelism

- ideally parallel execution is automated
- to this end ...
  - represent program as independent tasks
  - give some indication on the costs of tasks
  - don't specify number of threads

Futures

- like a function call
- returns a handle for the result
- may compute the function call in a separate thread
- synchronization upon accessing the result
- not yet in C++Ox working paper (i.e. example below is made up)

Future Example

```
result function(int arg) { .... }
int main()
{
  std::future<result> f1(std::bind(&function, 1));
  std::future<result> f2(std::bind(&function, 2));
  ...
  result const& r1(f1.get_result());
}
```

Parallel Algorithms

- Threading Building Blocks offers:
  - parallel_for(range, function [, splitter])
  - parallel_reduce(range, function [, splitter])
  - parallel_scan(range, function [, splitter])
- other algorithms can be built on top of these

General TBB Approach

- ranges splitable into equal-sized subranges
- ranges have an optional grainsize
- function objects splitable into multiple instances
- function objects responsible to process subranges
- subranges processed by separate threads

TBB Precondition

- every thread needs to initialize a scheduler
- ideally, this should be done at start-up
  - tbb::task_scheduler_init init(...);
  - in main() and thread entry function
- termination is at corresponding scope
- future: integrated into program/thread

tbb::parallel_for()

```
struct function { ...
  template <typename Range>
  void operator()(Range const& r) const {
    std::for_each(r.begin(), r.end(), *this);
  }
};
tbb::parallel_for(
  tbb::blocked_range<Iterator>(begin, end),
  function(...),
  tbb::auto_partitioner());
```

TBB Ranges

- ranges have grainsize, indicating atomic sizes
- grainsize may be heuristically determined
  - it isn’t exact
  - rule of thumb: at least 10,000 instructions
- have “split constructor”
  - take a dummy argument of type tbb::split

More On TBB Ranges

- tbb::blocked_range<T> works for
  - integral types
  - random access iterators
- ranges can be user defined
  - have to follow some simple concept

tbb::parallel_reduce()

- similar use to tbb::parallel_for()
- function object non-constant
- the result is accumulate using join() function
  on the function object
- used e.g. for find(), min_element(), etc.

tbb::parallel_scan()

- computes a scan for some associative $\oplus$
  - $y[0] = id \oplus x[0]$
  - $y[i] = y[i-1] \oplus x[i] \; \forall \; i > 0$
- two passes over each range
- dummy argument indicating which pass

Other TBB Compoents

- usual mutex, lock, etc.
- atomic operations
- concurrent containers
  - concurrent_queue<T>
  - concurrent_vector<T>
  - concurrent_hash_map<Key, Value,

TBB Summary

- restores free lunch when using many tasks
- library on which to build parallel components
- it probably needs some baking to work smoothly
- ... but it is certainly an interesting approach

Detailed Information

- Herb Sutter's articles in Dr.Dobb's which hopefully becomes "Effective Threading"
- "Programming with POSIX Threads", David R. Butenhof, Addison-Wesley
- "Intel Threading Building Blocks", James Reinders, O'Reilly
- www.sgi.com/tech/stl/thread_safety.html