Testing and Debugging Multithreaded Programs

CS3211 Parallel and Concurrent Programming

Outline

- Testing concurrent programs
- Debugging (multi-threaded) C++

Overview

- Testing and debugging concurrent code is *hard.*
- Any type of bug is possible in concurrent code
- Focus on concurrency-related bugs
  - Unwanted blocking – easier to discover
  - Race conditions – sometimes difficult to discover

Unwanted blocking

- Deadlock
- Livelock
- Blocking on I/O or other external input
  - Thread is blocked waiting for external input

Race conditions

- Data races
  - Observable: undefined behavior due to unsynchronized concurrent access to a shared memory location
- Broken invariants
  - Dangling pointers - another thread deleted the data being accessed
  - Random memory corruption - a thread reading inconsistent values resulting from partial updates
  - Double-free - two threads pop the same value from a queue, and so both delete some associated data
- Lifetime issues
  - The thread outlives the data that it accesses
    - Accessing data that has been deleted or otherwise destroyed
    - Potentially the storage is even reused for another object
  - Ensure that the call to `join()` can't be skipped if an exception is thrown
  - Example: thread references local variables that go out of scope before the thread function has completed

Techniques for locating concurrency-related bugs

- *Look at the code*
- *Testing*

Look at the code

- Many times you read what you intended to write, not what is written
- Get someone else to review
- Go through the code with a fine-tooth comb
- Take a break from the code, and return to it later
- Explain it to someone else (human or not)

Questions to use during reviewing code

- Which data needs to be protected from concurrent access?
- How do you ensure that the data is protected?
- Where in the code could other threads be at this time?
- Which mutexes does this thread hold?
- Which mutexes might other threads hold?
- Are there any ordering requirements between the operations done in this thread and those done in another? How are those requirements enforced?
- Is the data loaded by this thread still valid? Could it have been modified by other threads?
- If you assume that another thread could be modifying the data, what would that mean and how could you ensure that this never happens?

Testing multithreaded code

- Maybe the hardest type of testing
- Precise scheduling of the threads is indeterminate and may vary from run to run
  - The code does not *always* fail (Heisenbug)
  - Difficult to reproduce problems

Guidelines for testing

- Run the smallest amount of code that could potentially demonstrate a problem
  - Easier to locate the faulty code if the test fails
- Eliminate the concurrency from the test to verify that the problem is concurrency-related
  - For bugs found “in the wild”
- Run on a multicore and single-core your multithreaded code

Test various scenarios

For a queue:

- One thread calling push() or pop() on its own to verify that the queue works at a basic level
- One thread calling push() on an empty queue while another thread calls pop()
- Multiple threads calling push() on an empty queue
- Multiple threads calling push() on a full queue
- Multiple threads calling pop() on an empty queue
- Multiple threads calling pop() on a full queue
- Multiple threads calling pop() on a partially full queue with insufficient items for all threads
- Multiple threads calling push() while one thread calls pop() on an empty queue
- Multiple threads calling push() while one thread calls pop() on a full queue
- Multiple threads calling push() while multiple threads call pop() on an empty queue
- Multiple threads calling push() while multiple threads call pop() on a full queue

Test environments

- What you mean by “multiple threads” in each case (3, 4, 1024)?
- Whether there are enough processing cores in the system for each thread to run on its own core?
- Which processor architectures the tests should be run on?
- How you ensure suitable scheduling for the “while” parts of your tests?

Design for testability (in general)

- The responsibilities of each function and class are clear
- The functions are short and to the point
- Your tests can take complete control of the environment surrounding the code being tested
- The code that performs the particular operation being tested is close together rather than spread throughout the system
- You thought about how to test the code before you wrote it

Design for testability (concurrent code)

- Eliminate the concurrency - break down the code into
  - parts that operate on the communicated data within a single thread, and
  - parts responsible for the communication paths between threads
    - ensure that only one thread at a time *is* accessing a particular block of data
    - multiple blocks of *read shared data/transform data/update shared data* - testing the reading and updating of the shared data
- Watch out for library calls that use internal variables to store state
  - Use an alternate function safe for concurrent access from multiple threads

Techniques for multithreaded testing

- Stress testing (brute-force testing)
- Use special implementation of the library for synchronization primitives

Stress testing

- Run the code many times – with many threads running at once
- Write tests that maximize the problematic circumstances
- Platform to run the tests
  - Use multicore hardware
  - If available, multiple architectures
    - Example: on x86, load is the same independent of the memory ordering used
- Cover code paths

Special implementation of synchronization primitives library

- Mark your shared data in some way, and allow the library to check that operations on a particular shared data are done with a particular mutex locked
- Record the sequence of locks if more than one mutex is held by a particular thread at once
- Give priorities to threads to acquire a resource

Structuring multithreaded code

- Try to provide suitable scheduling for your threads to execute a particular part of the code

Testing the performance

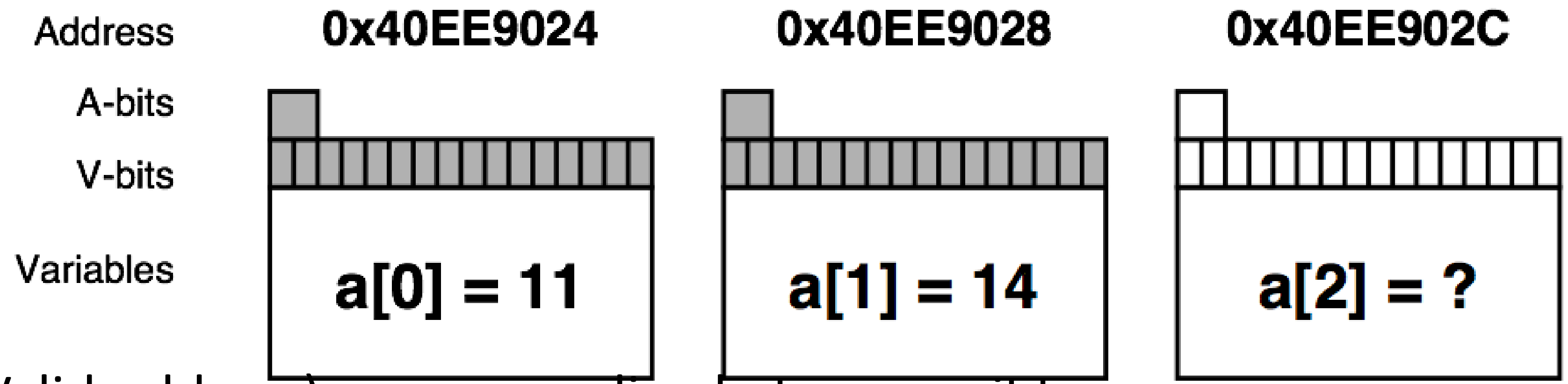
- **Scalability** – you want your code get the expected **speedup** when running with increasing number of threads on increasing number of cores
- Contention when accessing shared data from increasing number of threads

Debugging Tools

- Help in identifying (concurrency) bugs
  - Valgrind
    - Heavy-weight binary instrumentation: ~20x overhead
    - Designed to shadow all program values: registers and memory
      - Shadowing requires serializing threads
    - Thread error detectors: Helgrind, DRD
    - Dynamic instrumentations
  - Sanitizers
    - Compilation-based approach
- These tools need to keep track of the state of the memory
  - Compute *happens-before* to find data races

Valgrind memcheck – How to?

- Shadow memory
  - Used to track and store information on the memory that is used by a program during its execution.
  - Used to detect and report incorrect accesses of memory



A-bit (Valid-address): corresponding byte accessible

V-bit (Valid-value): corresponding bit initialized

Valgrind memcheck

- Validates memory operations in a program
  - Each allocation is freed only once
  - Each access is to a currently allocated space
  - All reads are to locations already written
  - 10 – 20x overhead

```
valgrind --tool=memcheck <prog …>
```

```
...
==29991== HEAP SUMMARY:
==29991==     in use at exit: 2,694,466,576 bytes in 2,596 blocks
==29991==   total heap usage: 16,106 allocs, 13,510 frees, 3,001,172,305 bytes allocated
==29991==
==29991== LEAK SUMMARY:
==29991==    definitely lost: 112 bytes in 1 blocks
==29991==    indirectly lost: 0 bytes in 0 blocks
==29991==      possibly lost: 7,340,200 bytes in 7 blocks
==29991==    still reachable: 2,687,126,264 bytes in 2,588 blocks
==29991==         suppressed: 0 bytes in 0 blocks
```

Helgrind in Valgrind

- Dynamic instrumentation
  - Intercepts calls to functions and instruments them
  - 100x slowdowns
- Detects:
  - Misuses of the POSIX pthreads API
    - By intercepting calls to many POSIX pthreads functions
  - Potential deadlocks arising from lock ordering problems.
  - Data races - accessing memory without adequate locking or synchronization.

Helgrind: Deadlock detection

- Helgrind builds a directed graph indicating the order in which locks have been acquired
- When a thread acquires a new lock
  - the graph is updated, and
  - then checked to see if it now contains a cycle.
- The presence of a cycle indicates a potential deadlock involving the locks in the cycle
- For 2 or multiple locks

```
Thread #6: lock order "0x80499A0 before 0x8049A00" violated

Observed (incorrect) order is: acquisition of lock at 0x8049A00
   at 0x40085BC: pthread_mutex_lock (hg_intercepts.c:495)
   by 0x80485B4: dine (tc14_laog_dinphils.c:18)
   by 0x400BDA4: mythread_wrapper (hg_intercepts.c:219)
   by 0x39B924: start_thread (pthread_create.c:297)
   by 0x2F107D: clone (clone.S:130)

 followed by a later acquisition of lock at 0x80499A0
   at 0x40085BC: pthread_mutex_lock (hg_intercepts.c:495)
```

Helgrind: Data races

- Check the order in which memory accesses can happen
  - *happens-before* relationship
  - builds a directed acyclic graph representing the collective happens-before dependencies
  - Monitors all memory accesses.
- No race in the case where both accesses are reads.

Happens-before in Helgrind

- When a mutex is unlocked by thread T1 and later (or immediately) locked by thread T2, then the memory accesses in T1 prior to the unlock must happen-before those in T2 after it acquires the lock.
- The same idea applies to reader-writer locks, although with some complication so as to allow correct handling of reads vs writes.
- When a condition variable (CV) is signalled on by thread T1 and some other thread T2 is thereby released from a wait on the same CV, then the memory accesses in T1 prior to the signaling must happen-before those in T2 after it returns from the wait. If no thread was waiting on the CV then there is no effect.
- If instead T1 broadcasts on a CV, then all of the waiting threads, rather than just one of them, acquire a happens-before dependency on the broadcasting thread at the point it did the broadcast.
- A thread T2 that continues after completing sem_wait on a semaphore that thread T1 posts on, acquires a happens-before dependence on the posting thread, a bit like dependencies caused mutex unlock-lock pairs. However, since a semaphore can be posted on many times, it is unspecified from which of the post calls the wait call gets its happens-before dependency.
- For a group of threads T1 .. Tn which arrive at a barrier and then move on, each thread after the call has a happens-after dependency from all threads before the barrier.
- A newly-created child thread acquires an initial happens-after dependency on the point where its parent created it. That is, all memory accesses performed by the parent prior to creating the child are regarded as happening-before all the accesses of the child.
- Similarly, when an exiting thread is reaped via a call to pthread_join, once the call returns, the reaping thread acquires a happens-after dependency relative to all memory accesses made by the exiting thread.

Sanitizers

- Compilation-based approach to detect issues
  - clang and gcc support
  - ~5-10x overhead
  - Add "-fsanitize=address", or "-fsanitize=thread", etc. for different sanitizers
- Examples:
  - AddressSanitizer (detects addressability issues) and LeakSanitizer (detects memory leaks)
  - **Thread sanitizers** (TSan)
  - MemorySanitizer (detects use of uninitialized memory)
  - HWASAN, or Hardware-assisted AddressSanitizer, a newer variant of AddressSanitizer that consumes much less memory
  - UBSan, or UndefinedBehaviorSanitizer

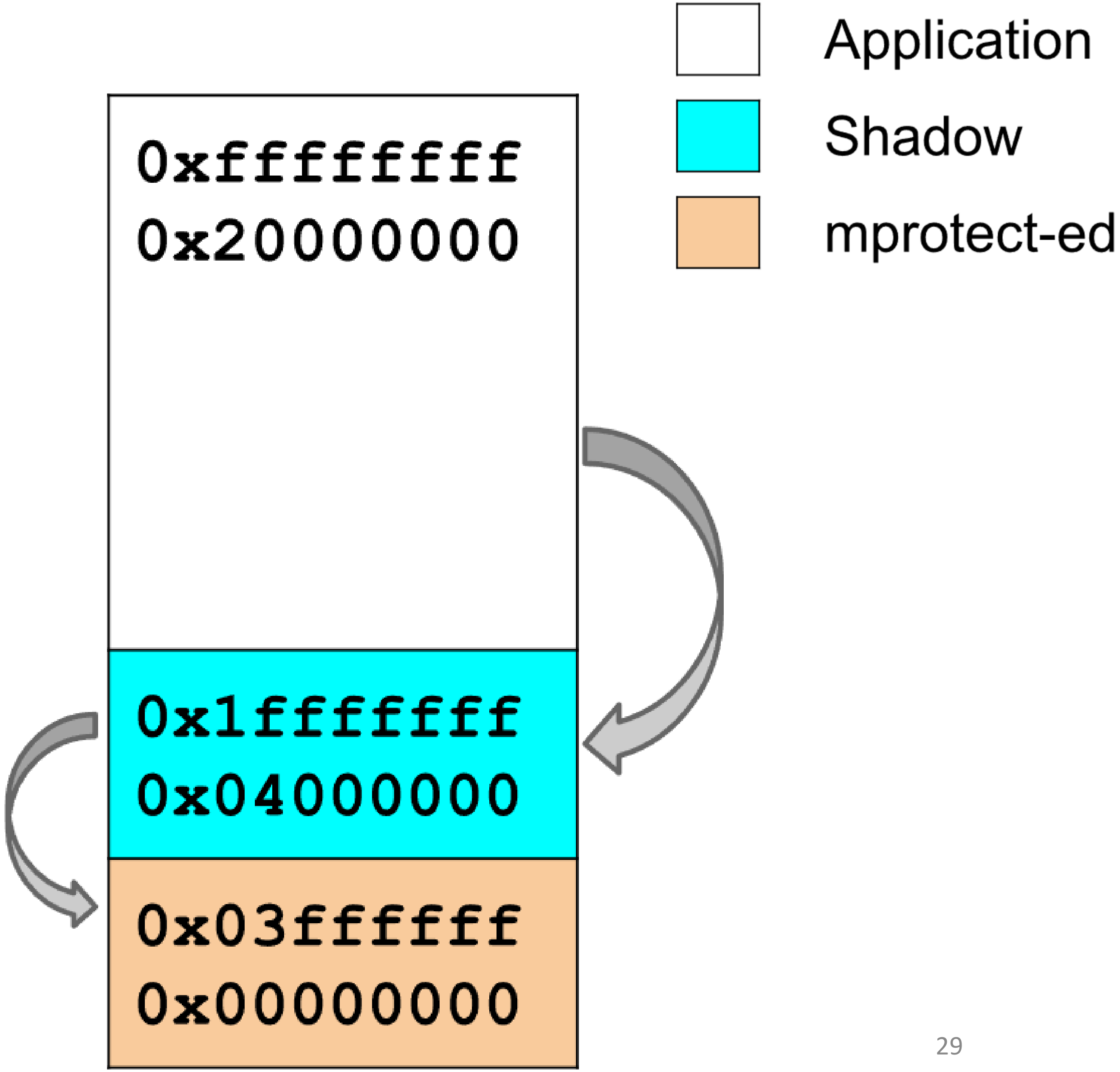
-fsanitize=address Example

```
int main(int argc, char **argv) {
   int *array = new int[100];
   delete [] array;
   return array[argc];  // BOOM
}
% clang++ -O1 -fsanitize=address a.cc && ./a.out

==30226== ERROR: AddressSanitizer heap-use-after-free
READ of size 4 at 0x7faa07fce084 thread T0
   #0 0x40433c in main a.cc:4
0x7faa07fce084 is located 4 bytes inside of 400-byte
region
freed by thread T0 here:
   #0 0x4058fd in operator delete[](void*) _asan_rtl_
   #1 0x404303 in main a.cc:3
previously allocated by thread T0 here:
   #0 0x405579 in operator new[](unsigned long)  asan rtl
```

Address Sanitizer (Asan)

- 2x slowdown
- 1.5-3x memory overhead
- Hundreds of bugs in Chrome
  - Used for tests and fuzzing



Asan – how it works?

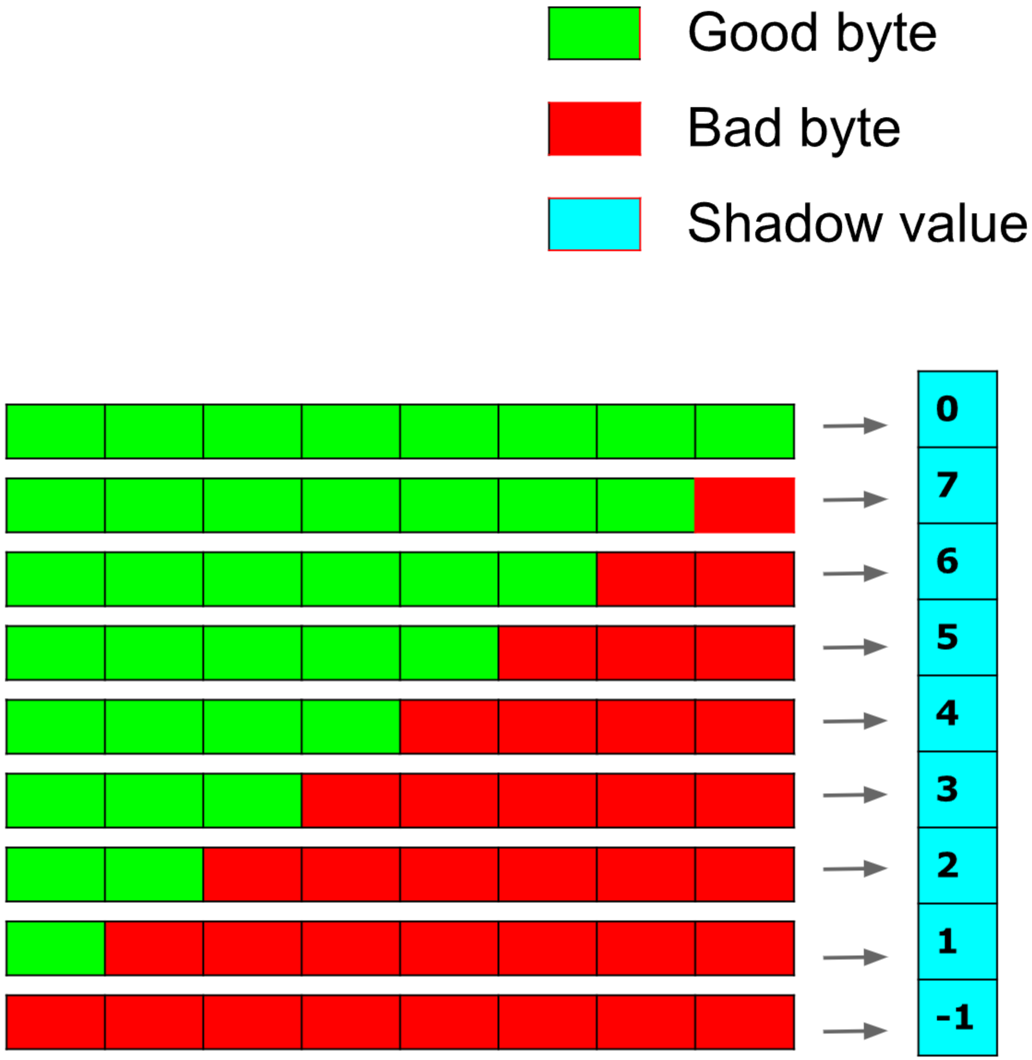
- Any aligned 8 bytes may have 9 states:

  N good bytes and 8-N bad (0<=N<=8)

- N byte access

```
*a = ...
```

```
char *shadow = addr >> 3;
if (*shadow &&
    *shadow <= ((a&7)+N-1))
  ReportError(a);
*a = ...
```



Thread Sanitizer (Tsan)

- Instrument a running program
- Engineered for speedup – 5-15x slowdown and about 5-10x memory overhead
- Discovered many races (hundreds) in Google server-side apps
  - Scales to huge apps
- Runtime library
  - Malloc replacement
  - Fully parallel
  - No expensive atomics/locks

Usage for Thread Sanitizer (TSan)

- Compile and link with `-fsanitize=thread, -fPIE, -pie`
  - Add –O2 for reasonable performance
- Run the executable (using options)
  - `TSAN_OPTIONS="history_size=7 force_seq_cst_atomics=1" ./myprogram`
- Tsan prints a report:

```
WARNING: ThreadSanitizer: thread leak (pid=9509)
  Thread T1 (tid=0, finished) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001fb33)
    #1 main thread_leak3.c:10 (exe+0x000000003c7e)
```

-fsanitize=thread Example

```c
#include <pthread.h>
#include <stdio.h>
int global;
void *Thread1(void *x) {
  global++;
  return NULL;
}
void *Thread2(void *x) {
  global--;

  return NULL;
}
int main() {
  pthread_t t[2];
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL);
  pthread_join(t[1], NULL);
}
```

```
==================
WARNING: ThreadSanitizer: data race (pid=17631)
  Read of size 4 at 0x562a4b132014 by thread T2:
    #0 Thread2 <null> (thread_san+0xa42)
    #1 <null> <null> (libtsan.so.0+0x296ad)
  Previous write of size 4 at 0x562a4b132014 by thread T1:
    #0 Thread1 <null> (thread_san+0xa03)
    #1 <null> <null> (libtsan.so.0+0x296ad)
  Location is global 'global' of size 4 at 0x562a4b132014
(thread_san+0x000000202014)
  Thread T2 (tid=17635, running) created by main thread at:
    #0 pthread_create <null> (libtsan.so.0+0x2bcee)
    #1 main <null> (thread_san+0xad3)
  Thread T1 (tid=17634, finished) created by main thread at:
    #0 pthread_create <null> (libtsan.so.0+0x2bcee)
    #1 main <null> (thread_san+0xab2)
SUMMARY: ThreadSanitizer: data race (.../thread_san+0xa42) in Thread2
==================
ThreadSanitizer: reported 1 warnings
```

Compiler-instrumentation with TSan

Instrumentation

- Function entry/exit
- Memory access

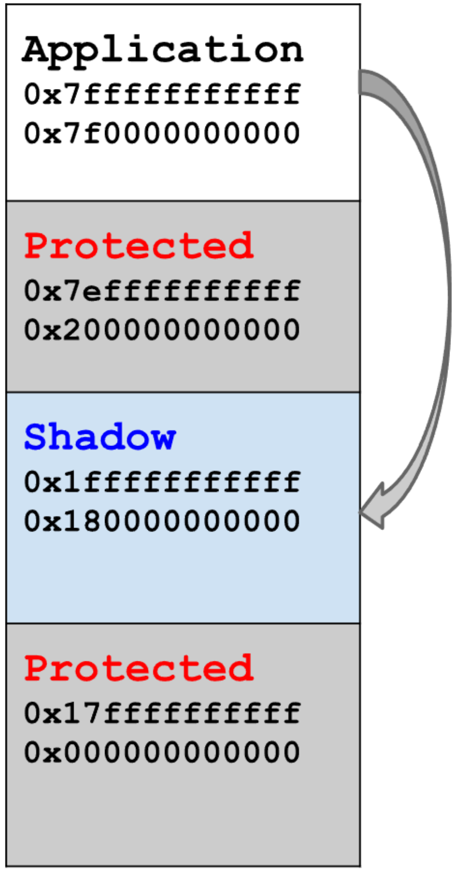
```
void foo(int *p) {
  *p = 42;
}
```

↓

```
void foo(int *p) {
  __tsan_func_entry(__builtin_return_addres
  __tsan_write4(p);
  *p = 42;
  __tsan_func_exit()
}
```

Direct shadow mapping

```
Shadow = 4 * (Addr & kMask);
```

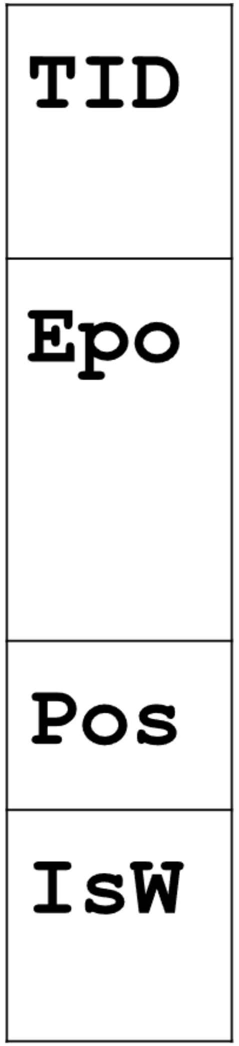


Shadow cell

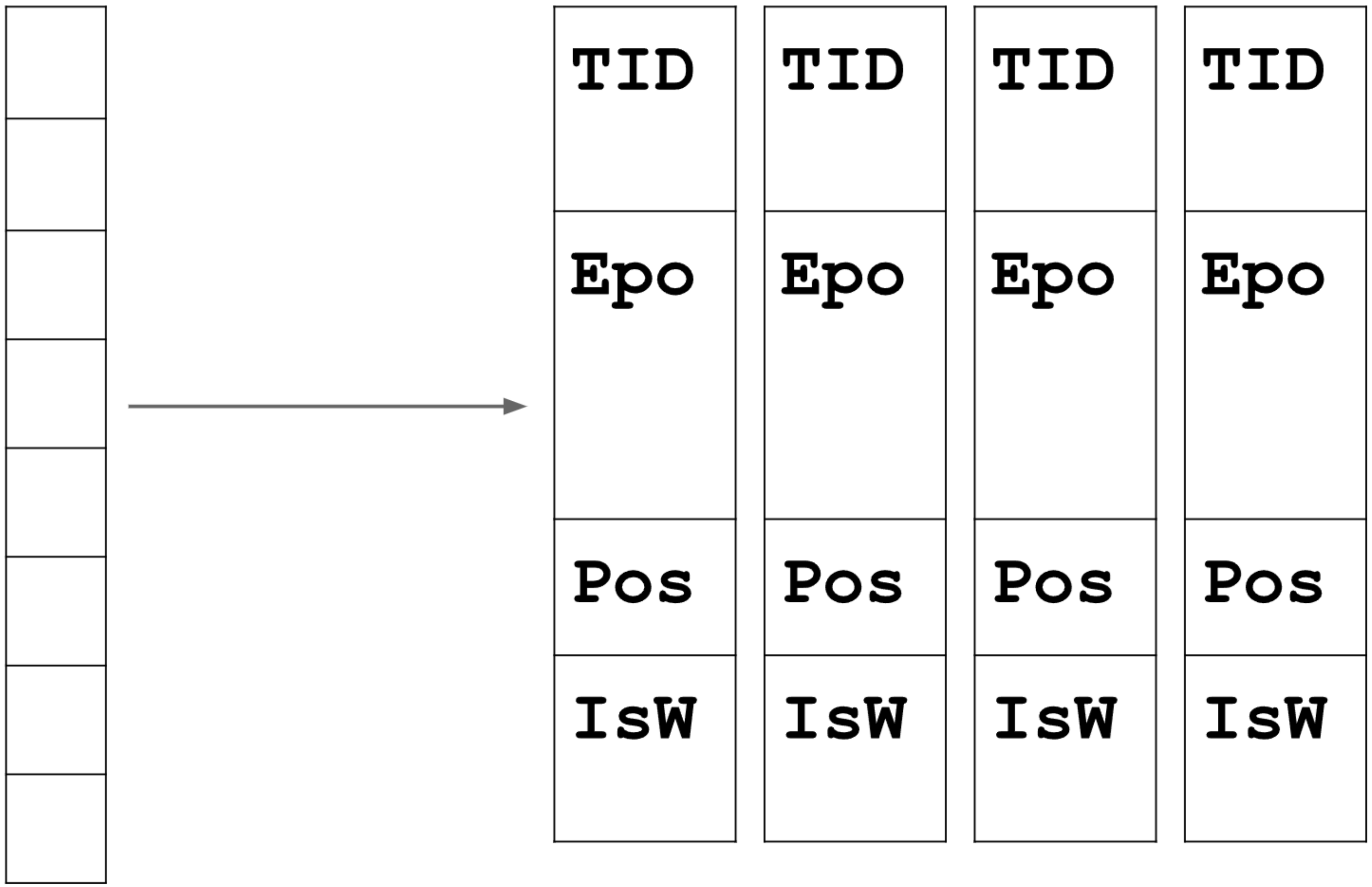
An 8-byte shadow cell represents one memory access:

- ~16 bits: TID (thread ID)
- ~42 bits: Epoch (scalar clock)
- 5 bits: position/size in 8-byte word
- 1 bit: IsWrite

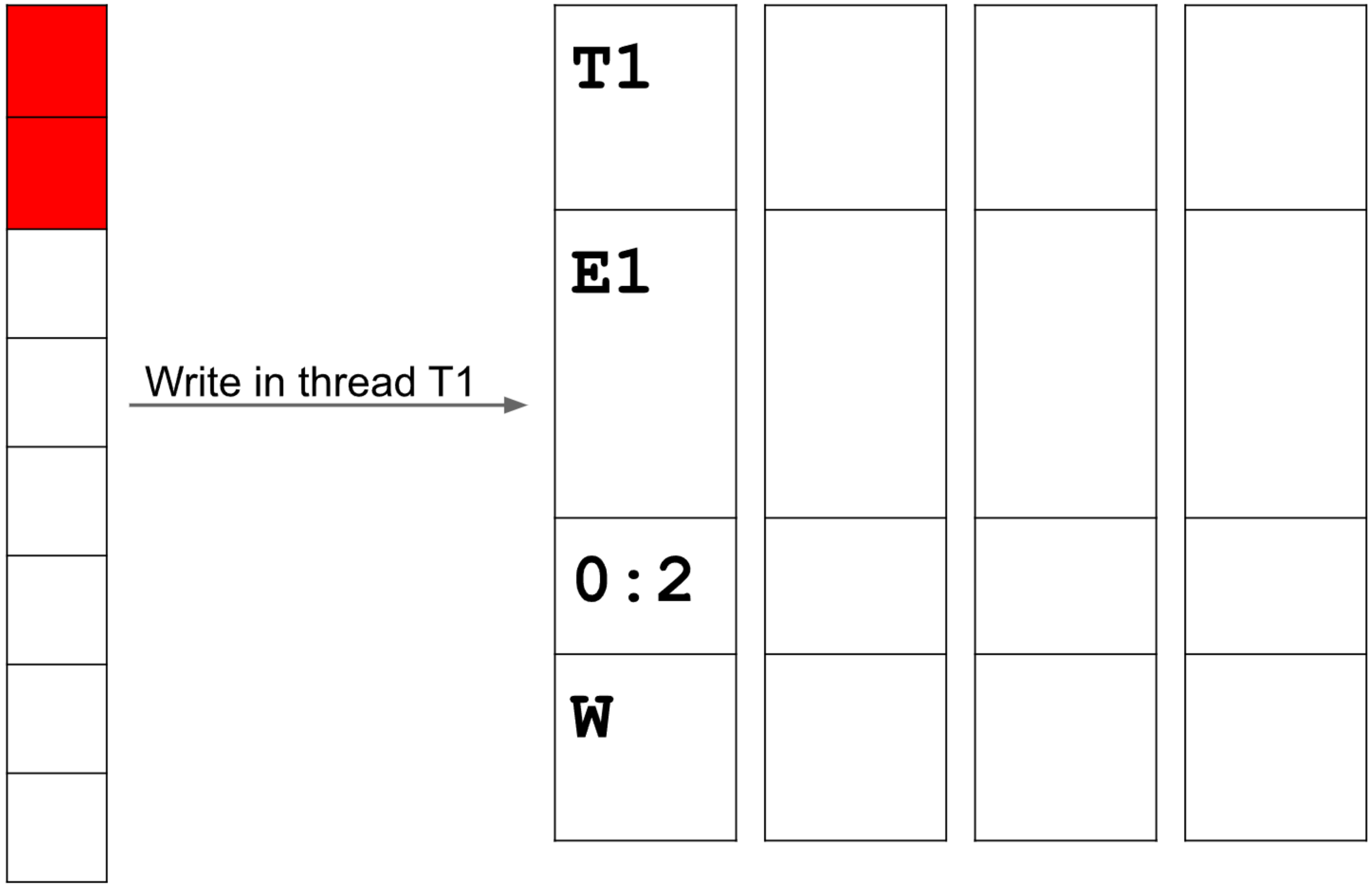
Full information (no more dereferences)



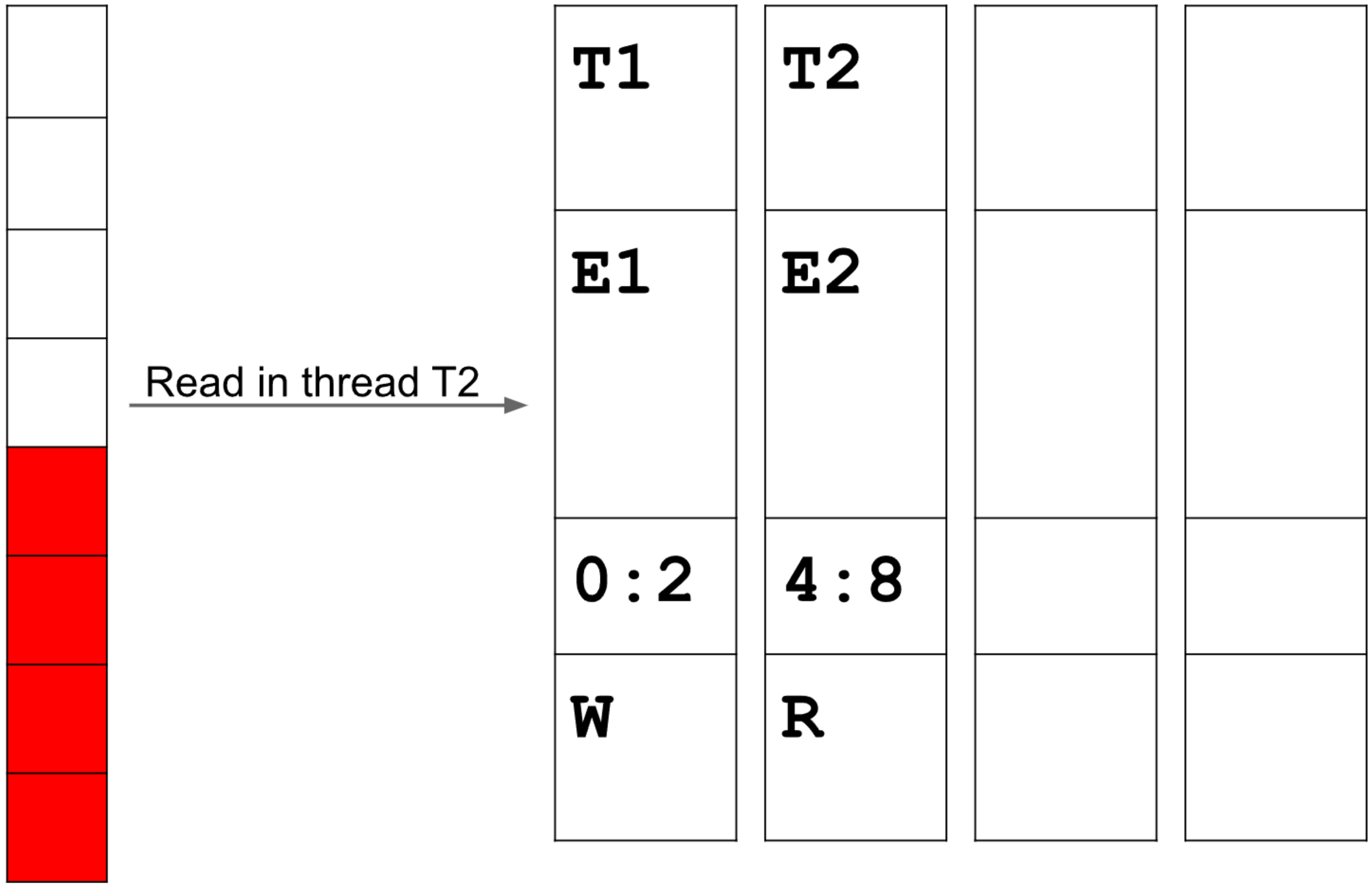
4 shadow cells per 8 application bytes



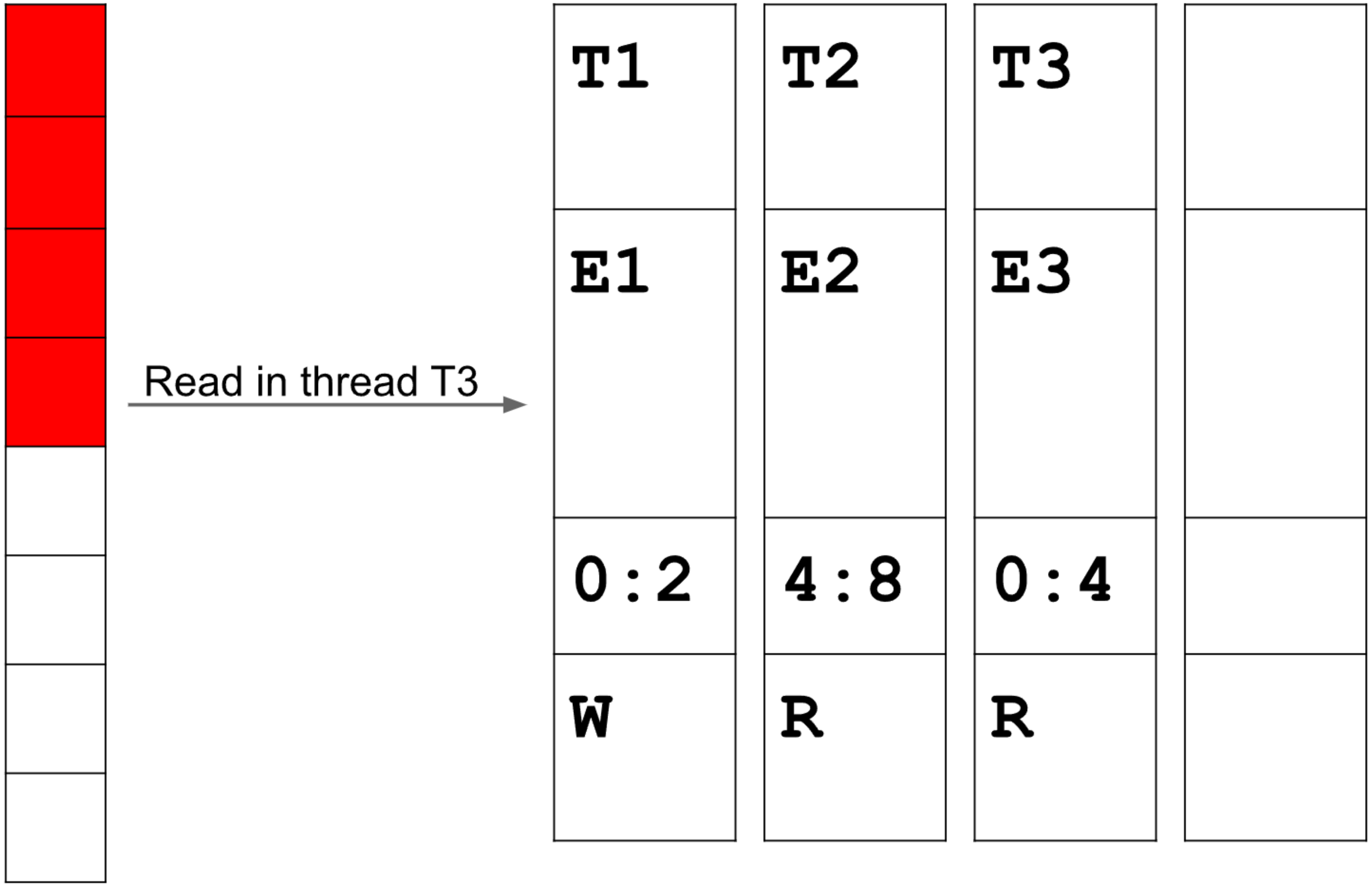
Example: First access



Example: Second access



Example: Third access



Example: Race?

- Race if E1 does not happen-before E3
- Constant time operation
  - Get TID and Epoch
  - 1 load from thread-local storage
  - 1 comparison

  (Details in FastTrack PLDI'09)

T1	T2	T3	
E1	E2	E3	
0:2	4:8	0:4	
W	R	R	

Stack trace for previous access

- Shown in the report from Tsan
- Per-thread cyclic buffer of events
  - 64 bits per event (type + PC)
  - Events: memory access, function entry/exit
  - Information will be lost after some time
  - Buffer size is configurable
- Replay the event buffer on report
  - Unlimited number of frames

Tsan detects

- Normal data races
- Races on C++ object vptr
- Use after free races
- Races on mutexes
- Races on file descriptors
- Races on pthread_barrier_t
- Destruction of a locked mutex
- Leaked threads
- Signal-unsafe malloc/free calls in signal handlers
- Signal handler spoils errno
- Potential deadlocks (lock order inversions)

* https://github.com/google/sanitizers/wiki/ThreadSanitizerPopularDataRaces

Action plan for debugging

1. Run Valgrind memcheck (and fix the errors)
2. Run Tsan (and fix the errors)
3. Run Asan (and fix the errors)
4. Run Helgrind (and fix the errors)

And suddenly C++ will become a safer language!

Summary

- Modern C++ is not easy, but has many concepts that are useful for us, as developers
- Concurrency in C++ is even more challenging
  - Built-in threads
  - Primitives for synchronization
  - Atomic<> Weapons
  - Testing & debugging
- Avoided C++ complicated syntax, as much as possible
  - Syntax is too “verbose” (according to our previous TA, Hao Wei)

References

- Chapter 11 from C++ Concurrency in Action
- Valgrind: https://valgrind.org/docs/manual/manual.html
- Sanitizers: https://docs.google.com/presentation/d/1LJ-MO0urqU__0id_WAfnfR9DI9gCJcZkkJwNxaCQW2k/pub?start=false&loop=false&delayms=3000&slide=id.gd300eb30_083