Hints for Writing Thread-Safe Code

Gene Cooperman
College of Computer and Information Science
Northeastern University, Boston, USA
gene@ccs.neu.edu

Threads Overview

- CPU speed, performance of one core no longer rising with Moore's Law
- Number of cores per CPU chip still rising approximately with Moore's Law: doubling every two years
- High-end "server-class' PCs (still under $10,000) typically have four CPUs on a motherboard.
- As of this writing, "google: Multi-core processor" reports: 6 cores per CPU chip commonly available: e.g. AMD Phenom II X6, Intel Core i7 Extreme Edition 980X
- Total: 24 cores on a PC, *and rising*

Scaling to Many Threads

- To gain full efficiency on 24 cores, you must have *at least 24 threads* today (in one process, or spread among multiple processes).
- *Multi-processing:* Multiple processes can also share data by using the Linux system call *fork* (using copy-on-write). However, a single write to a data page forces a private copy of that page. More cores places more pressure on cache and memory bus.
- One can fork a child process after the parent process initializes, to encourage maximal sharing of data. But there are problems:
  1. Some programs use lazy initialization.
  2. Many objects have both read-only fields *(after initialization)* and read-write fields. Examples of read-write fields occur due to caching of previous values, temporary intermediate values, re-use of objects to avoid allocation/freeing, etc.). A single write to a page can "poison" the performance by forcing a copy of that page.

Thread-safety

GOAL: Replace each process by a thread within a single process.

- Default for threads: All global data is shared.
- C keyword (also valid in GNU C++): `__thread`
  Any data declared with `__thread` is *thread-local* (non-shared).
- Keywords supported by existing compilers:
  GNU and Intel: __thread
  Microsoft C++: __declspec(thread)
- For C++, thread-local keyword will be standardized as:
  `thread_local` (upcoming "C++0x" standard)
- In C/C++, `__thread` / `thread_local` is permitted only for static data — *not* for fields of dynamically created objects. Geant4MT makes further transformations to guarantee dynamic data created by one thread is *thread-private* (a generalization of thread-local).
- Geant4MT transforms Geant4 so that most data is *thread-private*: data is thread-private if only one thread accesses that data.

Thread-safety (cont.)

Definition: A program routine is *thread-safe* if two threads can execute the same function at the same time, and the result is the same as if they had executed that function sequentially (one thread at a time).

Definition: A program is *pleasingly parallel* if all routines are thread-safe. (When possible, we would like all program routines to be pleasingly parallel.)

“Hello, world.” for Thread-Safe Code


```
// gcc -lpthread thisProgram.c; OR: g++ -lpthread thisProgram.cpp
#include <stdio.h>
#include <pthread.h>

#ifndef NUM_THREADS
# define NUM_THREADS 5
#endif

typedef struct { int thread_id; } thread_args_t;
__thread int thread_id = -1;  /* -1 means uninitialized */

void *thread_start(void *args) {
  thread_args_t *thread_args = (thread_args_t *)args;
  thread_id = thread_args->thread_id;
  printf("My thread id is: %d\n", thread_id);
  return NULL;
}
```

“Hello, world.” for Thread-Safe Code (cont.)


```
void *thread_start(void *);

int main() {
  pthread_t thread[NUM_THREADS];
  thread_args_t thread_args[NUM_THREADS];
  int i;

  for (i = 0; i < NUM_THREADS; i++) {
    thread_args[i].thread_id = i;
    pthread_create(&(thread[i]), NULL, thread_start, &(thread_args[i]));
  }

  for (i = 0; i < NUM_THREADS; i++) { /* Wait for threads to finish. */
    pthread_join(thread[i], NULL);
  }

  return 0;
}
```

Three Classic Synchronization Techniques

1. `pthread_mutex_lock()`, `pthread_mutex_unlock()` : *critical section* executed by at most one thread at a time
2. `pthread_rwlock_rdlock()`, `pthread_rwlock_wrlock` : either *multiple readers* or else *one writer* allowed in critical section, with *writer priority*.
3. producer-consumer : See
   `http://en.wikipedia.org/wiki/Producer-consumer_problem#Using_semaphores`
   for an example with working code.

Debugging


```
gcc -g -O0 -lpthread thisProgram.c
gdb ./a.out
(gdb) break 31
(gdb) run
Starting program: /home/gene/group/talks/geant4-thread-safe-11/a.out

[New Thread 0xb75a96d0 (LWP 12897)]
[New Thread 0xb75a8b70 (LWP 12906)]
My thread id is: 0
[New Thread 0xb6da7b70 (LWP 12907)]
My thread id is: 1
[Thread 0xb75a8b70 (LWP 12906) exited]
[Thread 0xb6da7b70 (LWP 12907) exited]
...

Breakpoint 1, main () at example.c:31
31   for (i = 0; i < NUM_THREADS; i++) { /* Wait for threads to finish. */
(gdb) info threads
* 1 Thread 0xb75a96d0 (LWP 12897)  main () at example.c:31
```

Debugging


```
(gdb) set scheduler-locking on
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/gene/group/talks/geant4-thread-safe-11/a.out
[New Thread 0xb76316d0 (LWP 12971)]
[New Thread 0xb7630b70 (LWP 12972)]
...

Breakpoint 1, main () at example.c:31
31   for (i = 0; i < NUM_THREADS; i++) { /* Wait for threads to finish. */
(gdb) info threads
  6 Thread 0xb562cb70 (LWP 12976)  0xffffe430 in __kernel_vsyscall ()
  5 Thread 0xb5e2db70 (LWP 12975)  0xffffe430 in __kernel_vsyscall ()
  4 Thread 0xb662eb70 (LWP 12974)  0xffffe430 in __kernel_vsyscall ()
  3 Thread 0xb6e2fb70 (LWP 12973)  0xffffe430 in __kernel_vsyscall ()
  2 Thread 0xb7630b70 (LWP 12972)  0xb7708658 in clone () ...
* 1 Thread 0xb76316d0 (LWP 12971)  main () at example.c:31
(gdb)
```

Examples of Code that is not Thread-Safe

1. Shared object with writeable field:
2. Example (race condition): Field of object that caches last computational result for re-use.
3. Example (atomicity): Code that keeps count of number of uses. FIX: Must replace by *atomic increment*: In GNU gcc-4.1 and later, use: GNU `__sync_fetch_and_add()`
4. *Dangers of a global variable that is writeable.*
   Depending on the code, some possible issues encountered are:
   (a) **Race condition:** Thread A and thread B race to write to a variable. Whoever writes last wins. Later results depend on who wrote last.
   (b) **Lack of bit compatibility:** Thread A *atomically* reads a global variable, adds a value x and writes it back. Thread B also *atomically* adds the value y to the same global variable. Hence, we either add $x+y$ or $y+x$ to the global variable. Unfortunately, on real computers, addition is not commutative in the least significant bits. The least significant bits are no longer reproducible.

Dangers of a global variable that is writeable.

4. *Dangers of a global variable that is writeable* . . . (cont.)

   (c) **Performance bug 1:** High-end motherboards support two or four CPU chips, but *without* cache on the motherboard. A write by one thread to a global variable must propagate to all CPU chips. This off-chip communication is slow. In one example that we encountered, verbose output was turned off for efficiency. But a "harmless" write to an associated shared global variable had not been turned off. This performance bug prevented scalability of Geant4MT beyond about 16 cores.

   (d) **Performance bug 2:** Every malloc package must include some variation on a central lock to handle the case when Thread A allocates memory and Thread B frees that memory. This is inherently non-scalable. Even the best of the newer malloc packages could not handle this.
   Solution: In most instances, when Thread A of Geant4MT allocates memory, Thread A is guaranteed to free it. For those instances, we provide a thread-private malloc arena (memory pool) for each thread of Geant4MT. This avoids the need for a central lock. We call this TPMalloc (Thread-Private Malloc).
   See: Xin Dong, Gene Cooperman and John Apostolakis, "Multithreaded Geant4: Semi-automatic Transformation into Scalable Thread-Parallel Software", *Proc. of Euro-Par 2010 — Parallel Processing*, Springer-Verlag, Lecture Notes in Computer Science **6272**, Springer, 2010, pp. 287–303

Combining Geant4MT with Sequential Software

Problems to watch out for:

- User-defined Hits, SteppingAction, Scorer, etc. may use code that is not thread-safe.
- Analysis packages are usually not multi-threaded. Multiple threads must hand off I/O for analysis to sequential code.

Example: When using Root, the information computed by an event is passed to a framework. Root processes that information while Geant4 simulates the next event. It is the responsibility of the framework to free the memory allocated by that first event. In this context, Geant4MT would require additional adaptation, such as creating a special “framework thread”. That framework thread would be responsible for passing the information of an event to Root, and later freeing the memory associated with that event.

Random Number Generators and Reproducibility

1. With multiple threads, each thread uses a separate random number generator.
2. To recover determinism (reproducibility), one must associate a unique random seed with each event number.
3. *Bit compatibility* (but see caveat below): As the number of threads increases each event number continues to produce the same results.

NOTE: For performance reasons, Geant4 will re-use previously computed results from an earlier event. In Geant4MT, a given thread "knows" only about earlier events computed by the *same* thread. (This is due to use of `__thread` keyword: thread-local data.) So, in order to guarantee full bit compatibility (including the least significant bits) between successive runs, one must provide a static thread schedule: each event number is associated not only with a unique random seed, but *also* with a unique thread. Thus, each event number always sees the same history of earlier events by that thread.

Tools from Geant4MT for Verifying Correctness

WARNING: Intended for Geant4MT developers (primarily for experts — please talk to us if you would like to use these tools)

Verifying comparable results between Geant4 and Geant4MT:

1. Bisimulation between Geant4 and Geant4MT with one thread
2. Geant4MT with one thread versus many threads (bit compatibility by design when using Geant4MT's static scheduling of threads)

(See talk by Xin Dong from Tuesday for details.)

Tools from Geant4MT for Verifying Correctness (cont.)

Verifying Geant4MT assumptions of read-only data after initialization: comparable results between Geant4 and Geant4MT

1. *Standard policy:* Remove write permission from memory pages (RAM) that is assumed read-only. If there is an attempt to write to read-only memory, Geant4MT will halt and report the instruction of code and the data address for the violation.
2. *Production run policy:* For added robustness during a production run, an alternative policy is possible. If a read-only memory violation is detected, halt all other threads, and temporarily grant write permission to the corresponding read-only memory. Then redo the corresponding event. Finally, again remove write permission and restart other threads at the beginning of their current event.

(See talk by Xin Dong from Tuesday for details.)