CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Introduction to Threads

- Multiple threads are often spawned to subdivide and collectively solve a larger problem.
- Consider the scenario where 10 ticket agents answer telephones—as they might have before the internet came along—at United Airlines to jointly sell 250 airline tickets.
- Each ticket agent answers the telephone, and each telephone call always leads to the sale of precisely one ticket.
- Rather than requiring each ticket agent sell 10% of the tickets, we'll account for the possibility that some ticket sales are more time consuming than others, some ticket agents need more time in between calls, etc. Instead, we'll require that all ticket agents keep answering calls and selling tickets until all have been sold.
- Here's our first stab at a **main** function. Full program is right here.

```
int main(int argc, const char *argv[]) {
  thread agents[10];
  size_t remainingTickets = 250;
  for (size_t i = 0; i < 10; i++)
    agents[i] = thread(ticketAgent, 101 + i, ref(remainingTickets));
  for (thread& agent: agents) agent.join();
  cout << "End of Business Day!" << endl;
  return 0;
}
```

Introduction to Threads

- As with most multithreaded programs, the main thread elects to spawn child threads to subdivide and collaboratively solve the full problem at hand.
- In this case, the **main** function declares the primary copy of the remaining ticket count—aptly named **remainingTickets**—and initializes it to 250.
- The main thread then spawns ten child threads to run some **ticketAgent** thread routine, yet to be fully defined. Each agent is assigned a unique id number between 101 and 110, inclusive, and a reference to **remainingTickets** is shared with each thread.
- As is typical, the **main** thread blocks until all child threads have finished before exiting. Otherwise, the entire process might be torn down even though some child threads haven't finished.

```
int main(int argc, const char *argv[]) {
  thread agents[10];
  size_t remainingTickets = 250;
  for (size_t i = 0; i < 10; i++)
    agents[i] = thread(ticketAgent, 101 + i, ref(remainingTickets));
  for (thread& agent: agents) agent.join();
  cout << "End of Business Day!" << endl;
  return 0;
}
```

Introduction to Threads

- The **ticketAgent** thread routine accepts an id number (used for logging purposes) and a reference to the **remainingTickets**.
- It continually polls **remainingTickets** to see if any tickets remain, and if so, proceeds to answer the phone, sell a ticket, and publish a little note about the ticket sale to **cout**.
- **handleCall**, **shouldTakeBreak**, and **takeBreak** are all in place to introduce short, random delays and guarantee that each test run is different than prior ones. Full program is (still) right here.

```
static void ticketAgent(size_t id, size_t& remainingTickets) {
  while (remainingTickets > 0) {
    handleCall(); // sleep for a small amount of time to emulate conversation time.
    remainingTickets--;
    cout << oslock << "Agent #" << id << " sold a ticket! (" << remainingTickets
         << " more to be sold)." << endl << osunlock;
    if (shouldTakeBreak()) // flip a biased coin
      takeBreak();         // if comes up heads, sleep for a random time to take a break
  }
  cout << oslock << "Agent #" << id << " notices all tickets are sold, and goes home!"
       << endl << osunlock;
}
```

Introduction to Threads

- Presented below right is the abbreviated output of a **confused-ticket-agents** run.
- In its current state, the program suffers from a serious race condition.
- Why? Because **remainingTickets > 0** test and **remainingTickets--** aren't guaranteed to execute within the same time slice.
- If a thread evaluates **remainingTickets > 0** to be **true** and commits to selling a ticket, the ticket might not be there by the time it executes the decrement. That's because the thread may be swapped off the CPU after the decision to sell but before the sale, and during the dead time, other threads—perhaps the nine others—all might get the CPU and do precisely the same thing.
- The solution? Ensure the decision to sell and the sale itself are executed without competition.

```
poohbear@myth61:$ ./confused-ticket-agents
Agent #110 sold a ticket! (249 more to be sold).
Agent #104 sold a ticket! (248 more to be sold).
Agent #106 sold a ticket! (247 more to be sold).
// some 245 lines omitted for brevity
Agent #107 sold a ticket! (1 more to be sold).
Agent #103 sold a ticket! (0 more to be sold).
Agent #105 notices all tickets are sold, and goes home!
Agent #104 notices all tickets are sold, and goes home!
Agent #108 sold a ticket! (4294967295 more to be sold).
Agent #106 sold a ticket! (4294967294 more to be sold).
Agent #102 sold a ticket! (4294967293 more to be sold).
Agent #101 sold a ticket! (4294967292 more to be sold).
// carries on for a very, very, very long time
```

Analysis of Ticket Agents Example

- Before we solve this problem, we should really understand why **remainingTickets--** itself isn't even thread-safe.
  - C++ statements aren't inherently atomic. Virtually all C++ statements—even ones as simple as **remainingTickets--**—compile to multiple assembly code instructions.
  - Assembly code instructions are atomic, but C++ statements are not.
  - **g++** on the myths compiles **remainingTickets--** to five assembly code instructions, as with:

```
0x0000000000401a9b <+36>:    mov    -0x20(%rbp),%rax
0x0000000000401a9f <+40>:    mov    (%rax),%eax
0x0000000000401aa1 <+42>:    lea    -0x1(%rax),%edx
0x0000000000401aa4 <+45>:    mov    -0x20(%rbp),%rax
0x0000000000401aa8 <+49>:    mov    %edx,(%rax)
```

  - The first two lines drill through the **ticketsRemaining** reference to load a copy of the **ticketsRemaining** held in **main** into **%rax**. The third line decrements that copy, and the last two write the decremented copy back to the **ticketsRemaining** variable held in **main**.

Improvements to Ticket Agents Example

- We need to guarantee that the code that tests for remaining tickets, the code that sells a ticket, and everything in between are executed as part of one large transaction, without interference from other threads. Restated, we must guarantee that at no other threads are permitted to even **examine** the value of **ticketsRemaining** if another thread is staged to modify it.
- One solution: provide a directive that allows a thread to ask that it not be swapped off the CPU while it's within a block of code that should be executed transactionally.
  - That, however, is not an option, and shouldn't be.
  - That would grant too much power to threads, which could abuse the option and block other threads from running for an indeterminate amount of time.
- The other option is to rely on a concurrency directive that can be used to prevent more than one thread from being anywhere in the same critical region at one time. That concurrency directive is the **mutex**, and in C++ it looks like this:

```
class mutex {
public:
  mutex();          // constructs the mutex to be in an unlocked state
  void lock();      // acquires the lock on the mutex, blocking until it's unlocked
  void unlock();    // releases the lock and wakes up another threads trying to lock it
};
```

Improvements to Ticket Agents Example

- The name **mutex** is a contraction of the words *mutual* and *exclusion*. It's so named because its primary use it to mark the boundaries of a critical region—that is, a stretch of code where at most one thread is permitted to be at any one moment.
  - Restated, a thread executing code within a critical region enjoys exclusive access.
- The constructor initializes the **mutex** to be in an unlocked state.
- The **lock** method will *eventually* acquire a lock on the **mutex**.
  - If the **mutex** is in an unlocked state, **lock** will lock it and return immediately.
  - If the **mutex** is in a locked state (presumably because another thread called **lock** but has yet to **unlock**), **lock** will pull the calling thread off the CPU and render it ineligible for processor time until it's notified the lock was released.
- The **unlock** method will release the lock on a mutex. The only thread qualified to release the lock on the **mutex** is the one that holds it.

```
class mutex {
public:
  mutex();          // constructs the mutex to be in an unlocked state
  void lock();      // acquires the lock on the mutex, blocking until it's unlocked
  void unlock();    // releases the lock and wakes up another threads trying to lock it
};
```

Final Solution to Ticket Agents Example

- We can declare a single **mutex** beside the declaration of **remainingTickets** in **main**, and we can use that **mutex** to mark the boundaries of the critical region.
- This requires the **mutex** also be shared by reference with the **ticketAgent** thread routine so that all child threads compete to acquire the same lock.
- The new **ticketAgent** thread routine looks like this:

```
static void ticketAgent(size_t id, size_t& remainingTickets, mutex& ticketsLock) {
  while (true) {
    ticketsLock.lock();
    if (remainingTickets == 0) break;
    handleCall();
    remainingTickets--;                                                      critical region
    cout << oslock << "Agent #" << id << " sold a ticket! (" << remainingTickets
         << " more to be sold)." << endl << osunlock;
    ticketsLock.unlock();
    if (shouldTakeBreak())
      takeBreak();
  }
  ticketsLock.unlock();
  cout << oslock << "Agent #" << id << " notices all tickets are sold, and goes home!"
       << endl << osunlock;
}
```

- When do you need a mutex?
  - When there are multiple threads **writing** to a variable.
  - When there is a thread **writing** and one or more threads **reading**
  - Why do you not need a mutex when there are no writers (only readers)?