Basic Threads Programming: Standards and Strategy

Mike Dahlin
dahlin@cs.utexas.edu

September 28, 2000

1 Advice and motivation

Some people rebel against coding standards. I don't understand the logic. For concurrent programming in particular, there are a few good solutions that have stood the test of time (and many unhappy people who have departed from these solutions.) For concurrent programming, *debugging won't work.* You must rely on (a) writing correct code and (b) writing code that you and others can read and understand. Following the rules below will help you write correct, readable code.

Rules 2 through 5 below are *required coding standards* for CS372. Answers to homework and exam problems and project code that do not follow these standards are by definition *incorrect.*

If you believe you have a better way to write concurrent programs, that's great! Bring it to us (before you use it on an assignment!) We will examine your approach the same way we hope that a civil engineering manager would examine a proposal by a bright young civil engineer who has a proposal for a better configuration of rebar for reinforcing a new bridge: we will hope you have found a better way, but the burden of proof for explaining the superiority of your approach and proving that there are no hidden pitfalls is on you.

Coding standards for concurrent programs:

1. Always do things the same way

   Even if one way is not better than another, following the same strategy every time (a) frees you to focus on the core problem because the details of the standard approach become a habit and (b) makes it easier for the people who follow you and have to review, maintain, or debug your code understand your code. (Often the person that has to debug the code is you!)

   Electricians follow standards for the colors of wire they use for different tasks. Black is neutral. Red or wite is hot. Copper is ground. An electrician doesn't have to decide "Hm. I have a bit more white on my belt today than black, should I use white or black for my grounds?" When an electrician walks into a room she wired last month, she doesn't have to spend 2 minutes trying to remember which color is which. If an electrician walks into a room she has never seen before, she can immediately figure out what the wiring is doing (wihout having to trace it back into the switchboard.) Similar advantages apply to following coding standards.

But for concurrency programs, the evidence is that in fact the abstractions we describe *are* better than almost all others. Until you become a *very* experienced concurrent programmer, you should take advantage of the hard-won experience of those that have come before you. Once you are a concurrency guru, you are welcome to invent a better mousetrap.

Sure, you can cut corners and occasionally save a line or two of typing by departing from the standards, but you'll have to spend a few minutes thinking to convince yourself that you are right on a case-by-case basis (and another few minutes typing comments to convince the next person to look at the code that you're right), and a few hours or weeks tracking down bugs when you're wrong. NOT WORTH IT!

2. Always use monitors (condition variables + locks) Either semaphores or monitors (condition variables + locks) could be used to write concurrent programs. We recommend that you be able to read semaphores (so you can understand legacy code), but that you only write new code using condition variables and locks (e.g., monitors.)

   99% of the time monitor code is more clear than the equivalent semaphore code because monitor code is more "self-documenting". If the monitor code is well-structured, it is usually clear what each synchronization action is doing. Sure, occasionally semaphores seem to fit what you are doing perfectly because you can map the invariants onto the internal state of the semaphore exactly (e.g., producer consumer), but what happens when the code changes a bit next month? Will the fit be as good? Rule #1 says that you should choose one of the two styles and stick with it, and my opinion is the right one to pick is monitors.

3. Always hold lock when operating on a condition variable

   You signal on a condition variable because you just got done manipulating shared state. You proceed when some condition about a shared state becomes true. Condition variables are useless without shared state and shared state is useless without holding a lock.

   Many modern libraries enforce this rule – you cannot call any condition variable methods unless you hold the corresponding lock. But some run-time systems you may see in the future allow sloppyness – don't fall for it.

4. Always grab lock at beginning of procedure and release it right before return

This is mainly an extension of rule #1 - pick one way of doing things and always follow it. (In particular, it is easy to read code and see where the lock is held and where it isn't because things break down on a proceedure by proceedure basis.)

Also, if there is a logical chunk of code that you can identify as a set of actions that require a lock, then that section should probably be its own procedure - it is a set of logically related actions. If you find yourself wanting to grab a lock in the middle of a procedure, that is usually a red flag that you should break the piece you are considering into a separate procedure. We are all sometimes lazy about creating new procedures when we should. Take advantage of this signal, and you will write clearer code.

5. Always use "`while(predicateOnStateVariables(...) == true/false){ condition->sleep()}`", not "`if...`"

   (Where `PredicateOnStateVariables(...)` looks at the state variables of the current object to decide if it is OK to proceed.)

   `While` works any time `if` does, and it works in situations when `if` doesn't. By rule 1, you should do things the same way every time.

   `If` breaks modularity. One might be tempted to use `if` if one convinced oneself that there will be exactly one signal when one waiting thread should proceed. The problem is that a change in code in one procedure (say, adding a `signal()`) can then cause a bug in another procedure (where the `wait()` is). `While` code is also self-documenting – one can look at the `wait()` and see exactly under what conditions a thread may proceed.

   When you always use `while`, you are given incredible freedom about where you put the `signals()`. In fact, `signal()` becomes a hint – you can add more signals to a correct program in arbitrary places and it remains a correct program!

I'm sure there are valid exceptions to all of the above rules, but they are few and far between. And the benefit you get by occasionally breaking the rules is unlikely to make up for the cost in your effort, extra debugging and maintenance cost, and loss of modularity.

2 Problem solving strategy

The following strategy should help you take a systematic approach to organizing your thoughts on a synchronization problem. Unlike the rules above, this is just advice. You are free to take any approach that works for you. Of course, if you write down your thinking at each of these steps, this is a good way to get lots of partial credit on exam questions.

1. Decompose problem into objects

- Identify units of concurrency. Make each a thread with a go() method. Write down the actions a thread takes at a high level.
- Identify shared chunks of state. Make each shared *thing* an object. Identify the methods on those objects – the high-level actions made by threads on these objects.
- Write down the high-level main loop of each thread.

Advice: stay high level here. Don't worry about synchronization yet. Let the objects do the work for you.

Now, for each object:

2. Write down the synchronization constraints on the solution. Identify the type of each constraint: *mutual exclusion* or *scheduling*
3. Create a lock or condition variable corresponding to each constraint
4. Write the methods, using locks and condition variables for coordination

Good luck.