The 12 Commandments of Synchronization

Emin Gün Sirer
Computer Science Department, Cornell University

October 4, 2011

Abstract

In the beginning, there was hardware. Now the hardware was formless and empty, darkness was over the surface of silicon.

And then the creator said "let there be operating systems," and there were OSes. The creator saw that OSes were good.

And the creator said "let there be processes, and threads." OSes were teeming with processes and threads carrying out different tasks. Then the creator said "let the processes and threads synchronize with each other." For this task, the creator appointed human-kind. But humans were fallible, and weak, and they failed to get synchronization correct, and fallen angel BSOD (pronounced *beesod*), spawn of Beelzebub, ruled the day with great evil.

So the creator sent the following commandments.

Commandment 0. *Thou shalt live and die by coding conventions for synchronization.*

Remember the parable of Jebediah the Electrician, who wired up his house using only pink wires. He said onto the townsfolks "I saved a bundle by not buying properly color-coded wires; and yet my lights turn on when I flippeth the switch. My wiring is correct and just right for my needs, and I did not have to worry about blue wires and green wires and brown wires. I use the book of Good Wiring Conventions as TP in my outhouse, for I have no use for any rules." And the townsfolks said "be careful with your hubris, for the creator might smite thee." And he respondeth, "I rule over you with my cheap wiring and you are all weaklings for following well-known good practices." Remember when his wife was with child, and he wanted to build an addition to his outhouse? He made a mistake when tracing the all-pink wires all the way back to the power plant, cut into a live wire with his metal clippers, and was promptly fried in a flash of hellfire. Take heed of the story of ex-Jebediah the Electrician, now known as Jebediah the Crisp.

For that is the fate that awaits you when you do not follow conventions when writing synchronization code. There will be times, tempting times, when the problem seems constrained enough to solve with a custom solution that violates the conventions you have been taught. We sent these to you to test your free will and self-control, and BSOD lurks behind them. If you give in to the temptation, you will come up with a solution that may be correct just by the skin of its teeth, but will likely not be understandable to others in the field. It may be just right, or it may have subtle bugs, but in either case, it will take a lot of time to figure out whether it is correct. Remember that your goal is not to barely meet the minimum correctness criteria—we created you in our image to strive for higher standards. Your goal is to meet the correctness criteria with code that is clear, obviously correct, self-documenting, and maintainable. So for that reason, always follow the commandments, even when not doing so does not immediately lead to problems that you yourself can observe. For at a minimum, you make it difficult for the grader angels to track your sins and good deeds!

And surely, the grader angels have better things to do than to prove your code correct or incorrect. Just as they do not spend their time listening to every utterance by every crazy person to see if there is any grain of truth in their monologues, they cannot be expected to find the nugget of goodness in a sea of jumbled synchronization code. They need to see clear, crisp, obviously correct code. The onus is on you to not just write correct code, but to demonstrate to others easily that it is correct. And we have given you these commandments, and the dragon book, and the suplemental reading list, and the miniprojects, and the lectures, so you may do so.

Commandment 1. *Thou shalt name your synchronization variables properly.*

We gave you all names, and so should you to all your variables. And the name of each object should describe that which the object does. Who amongst you would name your kid "Kid"? Or "KidA"? Or "MyKid"? Or "k"? Every kid is plainly a kid, and someone's kid. A global named `lock` or `mylock` or `mutex` or `sema` or `l` is an abomination. For a big program will have countless mutexes for different pieces of functionality, so you shall, nay, must, document what exactly that particular mutex is trying to achieve. For we watch over your code, and we can see the line `sema = Semaphore()` so we know that `sema` is, indeed, a Semaphore without your helpful(!) naming.

The name has to describe the function the variable serves. If it's used to provide mutually exclusive access to a resource R, call it `R_mutex`. If it's a semaphore used to count the number of free slots in a bounded buffer, call it `freeslots`. If it's a condition variable where the barber waits for customers to enter, you may call it `customerspresent`, so when you write code like `customerspresent.wait()`, the code is self-documenting. If it's used to ensure mutually exclusive access to an object, and it is an instance variable of that object, then, and only then, may you call it `lock` and refer to it as `self.lock`—it describeth that which it locks, because it is an instance variable (and not a global). Only those who have been corrupted by the great evil of BSOD would expect someone to follow synchronization code where the variable names consist of a single letter like `s`, or worse, where multiple synchronization variables have names that differ in just a few characters, like `sema1` and `sema2`. For that is surely the road to synchronization hell. All it takes is a trickster djinn or Nanabozho to cause a typo, and BSOD shall rule.

Commandment 2. *Thou shalt not violate the abstraction boundaries provided by synchronization primitives, nor shalt thou try to change the semantics of well-established synchronization primitives, and thou shall look with disdain upon he who does.*

Semaphores solely support an init()/P()/V() interface, with no way to broadcast() or notifyAll(), and permit no way to read the internal count. Condition variables provide wait()/signal() and broadcast(), but they have no internal state, and they provide no way to tell if there are sleeping threads on the condition variable. There is no way to tell if a monitor lock is currently being held without entering the monitor.

There are dangerous lands where basic synchronization primitives provide additional interfaces. For instance, the tongue of Python allows you to check if a lock is held without acquiring it. Such extended interfaces are the work of BSOD, for they are worse than crack cocaine. They are tempting to use, but will likely lead to the curses of busy-waiting or incorrect synchronization.

As for people who invent their own new synchronization mechanisms that are neither semaphores nor monitors with condition variables. Recall the parable of John Doe the Microsoft Programmer who implemented the process suspend counts in Windows. These counts count down, but not up, so they're half-semaphore, and half-condition-variable. In the Land of Windows, if you do a suspend() followed by another suspend() on a thread, you must perform two resume() operations to render the thread runnable. But if you perform two resume() operations on a running thread followed by a single suspend(), your thread will be suspended. And if you perform two resume()s followed by two suspend()s, then you still have to perform two more resume()'s to get the thread runnable again, even though, often, you have no idea when your resume() operations are scheduled to execute. There is no way to convey the amount of pain that this asymmetry has caused, except to say that there is a lesser-known 8th level of hell in Dante's Inferno reserved, solely, for John Doe and his ilk.

Do not invent your own synchronization mechanisms, for you shall only come up with unnatural abominations of the kind used in Windows, resembling the half-man, half-bear and half-pig we materialized to strike terror in people's hearts. The existing primitives are universal and sufficient. Master them and you shall need additional features in your synchronization mechanisms about as much as you need a second tail. "Wait," you might say, "I don't even have a first tail." That is precisely the point.

Commandment 3. *Thou shalt use monitors and condition variables instead of semaphores whenever possible.*

Synchronization bugs started the day Lady Ada, the first programmer, traveled through time and bit into the apple offered by Edsger Dijkstra, which contained Semaphores. Semaphores, as powerful and useful as they are, are too powerful for fallible humankind—recall the parable of Gollum, who was originally an IBM 360 programmer. You should be sufficiently conversant in semaphores, so you can speak to the lost tribes who speak in that tongue. You might find yourself in semaphore-only runtimes, where you have no choice but to write semaphore code.

But we gave you monitors and condition variables so you do not have to worship the ancient gods of semaphores! You know that monitors and condition variables are designed to make the code self-documenting. You know that they make explicit the condition for which a thread is waiting, without having to read through the totality of the code, as would be the case if semaphores were used. You know that they separate the mutual exclusion functionality of binary semaphores from the wait/signal functionality provided by counting semaphores into monitors and condition variables, respectively. Given all these advantages, you should always default to a solution based on monitors and condition variables.

The tongue of Python is the closest one to the original vision of the creator. There, you define a monitor with condition variables thus:

```
class BoundedBuffer:
    def __init__(self, maxsize):
        self.lock = Lock()
        self.maxsize = maxsize
        self.items = Queue()
        self.itemspresent = Condition(self.lock)
        self.freespace = Condition(self.lock)

    def produce(self, item):
        with self.lock:
            while len(self.items) == self.maxsize:
                self.freespace.wait()
            self.items.append(item)
            self.itemspresent.notify()

    def consume(self):
        with self.lock:
            while len(self.items) == 0:
                self.itemspresent.wait()
            self.freespace.notify()
            return self.items.pop()
```

Note how the condition variables are all wrapped around the same monitor lock. This is what ties the entire monitor together, as this is how the system figures out which monitor lock to release when the thread blocks on a condition variable. It is incidentally the same monitor lock you should be holding at the time you invoke operations on condition variables. If you omit that part, and create condition variables thus `Condition()`, the condition variables shall have a brand new Lock() instance associated with them, and you shall have to acquire that particular lock in addition to the monitor lock. The creator looks down on such behavior, not because it is inefficient, but because it is confusing, and all good hackers should stay away from confusing code.

Note also the `with` statements that acquire and release the monitor lock on entry and exit to the monitor-related procedures. Note also that there is only one lock per monitor. BSOD might lure you with promises of even greater concurrency and tempt you into using more than one lock per monitor. Do not give in, for the number of the beast, when it comes to monitor locks, is every number greater than one.

Commandment 4. *Thou shalt not mix semaphores and condition variables.*

You belong to the creator's chosen tribe of hackers who can write correct synchronization code. While you need to understand semaphores so you can traverse the land of the infidels who use them, you shall not adopt their customs, or mix and match their customs with ours. Such inter-species couplings are certainly bound for hellfire.

Commandment 5. *Thou shalt not busy-wait.*

You are reading this note because you want to figure out how to use blocking synchronization primitives properly. So you must understand the need to avoid spinning. Be careful of BSOD and his ways of getting you to spin even though you're ostensibly using blocking primitives. You might find yourself in a while loop, constantly grabbing a lock, checking a predicate, and if the predicate does not hold, relinquishing the lock and going back to the beginning of the loop—this is a busy-waiting loop even though there is a lock or semaphore involved. A loose indicator of such bad code is when you find yourself wanting to put a sleep() or yield() statement in the loop to "give another thread a chance to run." Another loose indicator of a busy-wait is when your laptop is too hot to hold in your lap. A better indicator is to ask the question, for each thread, under what conditions that thread will ever block and will be taken off the run queue. You must have a concise, non-empty answer to this question.

Commandment 6. *All shared state must be protected.*

Thou shalt protect all data that may be accessed by more than one thread. A simple way to check proper protection is to ask the question, for each variable or each object allocated on the heap, what is the set of locks (monitors or semaphores) that are held at the time that piece of memory is touched. This is known as the Lock-Set check. Compute the Lock-Set of each variable X at every point where it is read or written. Take the intersection of all Lock-Sets. The result must be non-empty, or else BSOD will smite your program at a time when you expect it least. And don't forget that both reads and writes must happen under the same lock set; reads cannot be performed without a lock, or else they may observe an inconsistent state.

Do not get complacent because updates to integers are atomic on most architectures! The creator does not guarantee atomicity of multiple operations on multiple integer fields, of operations on data structures such as queues and trees and hashtables, or on some architectures, of accesses to plain old integers. (On some high-performance multiprocessors, even though writes to integer variables are atomic on any given processor, these writes are not visible to other processors unless the processors perform a "memory barrier," a special instruction that flushes the processor's write buffers. On these machines, you must judiciously issue memory barriers before and after touching shared state. Locks are the simplest way to ensure that appropriate memory barriers are issued on these architectures).

If any variable is accessed without the corresponding lock, the reader of your code will immediately conclude that you're an unknowning disciple of the infidel BSOD, unless you also include a large comment that explains, in great detail, why you've *had to* violate this commandment. Also, if you have had to violate this commandment, you have to shout "unclean" as you roam the countryside, so others know that your wretched code is diseased, and shall treat it carefully.

Commandment 7. *Thou shalt grab the monitor lock upon entry to, and release it upon exit from, a procedure.*

While BSOD and his ilk will try to convince you to hold locks for as short a time as possible, do not ever forget that your primary allegiance is to correctness. If you grab and release the monitor lock more than once in your program, the monitored state can change in between, which creates the opportunity for bugs to creep in. Therefore, you must grab the monitor lock on entry, immediately, and make sure to release it on all return paths, but only at the very end.

The right way to do this in the tongue of Python is to use the `with` statement as the first line in every procedure that is part of the monitor. In Java, the `synchronized` keyword on a method ensures that the monitor lock is acquired and released automatically, but thou shall stay away from its sinner cousin, the `synchronized`-block within a method, for he shall lead you astray.

Commandment 8. *Honor thy shared data with an invariant, which your code may assume holds when a lock is successfully acquired and your code must make true before the lock is released.*

Remember that you created your monitor to protect some shared state from conflicting updates. That state shall have placed upon it an invariant. Honor this invariant, so that the days of operation of your program are long. The nature of the invariant is specific to the problem at hand. Invariants may include the myriad of properties with which the creator has blessed programs, such as the correspondance between values in counters and various entities in the world, correspondance between state variables and the history of past accesses to the shared state, and statements pertaining to the size or structure of any data structures. Typically, the particular invariant for your shared state will be tightly coupled to or derived from the safety (and perhaps liveness) properties of the problem you are trying to solve. For instance, in the bounded buffer example of which the creator spoketh, the invariant is that the queue holds between 0 and `maxsize` items.

Enunciate the invariant, so you might uphold it upon monitor creation. Ideally, write it down in the monitor creation code, so those who follow in your footsteps will know to uphold it. Having enunciated an invariant, and made sure it holds upon monitor creation, you may assume that it holds on any entry to the monitor. Assumptions are the reward that you receive for following this commandment, for they allow you to make progress on writing code: where would we be if we could not make assumptions about the state of our variables? But you must pay for the privilege of making assumptions about shared state with a corresponding obligation, for the creator both giveth and taketh. And your obligation is to ensure that the same invariant, which the creator allowed you to freely assume upon entry to your monitor, is upheld at the time the monitor lock is released. Thus, honor the invariant, and it shall honor you.

Commandment 9. *Thou shalt cover thy naked waits.*

Condition variables are not semaphores for a reason. Their usage is supposed to convey the precise condition for which the thread is waiting. Whereas for semaphores, figuring out this condition requires analyzing the rest of your program, condition variables properly used can make this condition apparent to every reader. This requires a pattern where the wait is preceded by a predicate. You must always have such a predicate. Is it possible to sometimes get away without such a predicate and just issue a "self.cv.wait()" without checking any state? Maybe, but recall the fable of Jebediah the Crisp, who cut corners. So, in this course and also later in life, you must always precede every wait on a condition variable with an appropriate check for the condition on which that thread is blocked.

And don't try to appease this commandment by adding a gratuitous predicate! If you replace `self.cv.wait()` with

```
with self.lock:
    a=False
    while not a:
        self.cv.wait()
        a=True
```

your wait is just as naked as before. The creator sees through all such superficial makeup on thy nakedness. The predicate must refer to variables that are shared with, and modified by, other threads.

Commandment 10. *Thou shalt guard your wait predicates in a while loop. Thou shalt never guard a wait statement with an if statement.*

When the creator first imagined the universe, he thought of a world where a signal immediately passed the monitor lock from the signaler to the signalee. This design was ok, and was called Hoare semantics, but it had the property that it interfered with scheduling.

Then some brilliant practitioners from the tribe of Mesa had the idea to not immediately schedule the signalee and hand over the monitor lock at the time of the signal. In the tribe of Mesa, the signalee simply gets placed on the run queue, with no guarantees on when it will grab the monitor lock. It may have been woken up because there is an item to consume, but then some other thread may have entered the monitor and consumed the item before our original thread got a chance to execute. Therefore, upon every wakeup, the predicate for waiting needs to be rechecked.

To do this, you must use the idiom

```
with self.lock:
    while not wakeup-condition
        self.cv.wait()
```

Note the while loop and the predicate, and how every wakeup from wait is immediately followed by re-checking of the condition.

Commandment 11. *Thou shalt not split predicates.*

Sometimes, you might find yourself wanting to sleep on two separate condition variables, depending on a conditional check. If you find yourself having to wait for two predicates to become true, you must check both of them at the same time. By the same time, we mean while holding the same monitor lock during the same epoch. Assume that a thread grabs a monitor lock, checks a predicate A, waits (by releasing the monitor lock) until A is true, wakes up, checks A and finds it true, checks B, and waits (by releasing the monitor lock) until B is true. When it finally wakes up and re-checks B, it can at most be assured that B holds. A may have changed to be false while the monitor lock was released, waiting for B to turn true. So the following code snippets are very much *not* equivalent.

This is the path of the righteous one:

```
with self.lock:
  while not self.condA or not self.condB:
     if not condA:
       self.condA_cv.wait()
     if not condB:
       self.condB_cv.wait()
  # here, both condA and condB hold
```

Note that on wakeup from either the first or second condition, both conditions are re-checked, as they should be.

And this is the path of the infidel, leading to the depths of hell:

```
with self.lock:
  while not self.condA:
    self.condA_cv.wait()
  while not self.condB:
    self.condB_cv.wait()
  # here, condA does not necessarily hold
```

Commandment 12. *Thou shalt help make the world a better place for the creator's mighty synchronization vision.*

There are many systems, and great systems yet to come, that provide synchronization primitives. And while the creator sent many prophets to preach to humankind, some wicked practitioners paid no heed and did not follow his word. For instance, some languages provide at most one condition variable per monitor. C++ provides no special language support for monitors and condition variables, only runtime libraries. Some languages lack synchronization support altogether. May the creator have mercy on their souls.

Once you have mastered the basic primitives, you have a life-long obligation to help those who have not. Do guide your brother and sister programmers educated elsewhere through the valley of their limited synchronization APIs to a modern future that lives up to the creator's vision, for you are their keeper. Pay no attention to followers of BSOD who preach that synchronization is too difficult for mortals; they peddle their event-programming frameworks, which are slow and work only at small scales. Such frameworks lack the power, concurrency and flexibility of the multi-programming model with which the creator has blessed you. You do not need someone else's framework to grab a single, global lock for you. Show that you're capable of handling concurrency, and the creator shall further bless you with correctly working code.

Once you've mastered these basics, you are destined for Heaven, where you will meet the creator, get 52 processor cores all to yourself, and get to break every commandment herein for eternity. There is an exciting research area, involving non-blocking (lock-free) synchronization primitives, where a select group of people who have mastered the teachings in these commandments play an even higher-performance game, and none of these commandments are used at all. But getting there requires a mind-merge with the creator, which starts by following these commandments religiously.

May you live long, build complex yet correct software, and may our breed of hackers who can write correct synchronization code prosper.

Bibliography

Inspiration for this document came from these holy texts that preceded it:

1. Mike Dahlin. Basic Threads Programming: Standards and Strategy. Department of Computer Sciences, University of Texas at Austin, Tech Report TR-08-07, 2008

2. Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. Communications of the ACM, 23(2):105-117, 1980.

3. Fred B. Schneider. On Concurrent Programming. Springer-Verlag, New York, New York, 1997.

4. Andrew Birrel. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.

5. C. A. R. Hoare. Monitors: an operating system structuring concept. Communications of the ACM, 17(10):54957, 1974.

6. Edsger W. Dijkstra. Cooperating sequential processes (EWD-123). E.W. Dijkstra Archive. Center for American History, The University of Texas at Austin, September 1965.