Matt Stewart
V00218956

CSC 464: Assignment 1

Classical (and Not-So-Classical) Synchronization Problems

Introduction

The following are simple solutions for some classical, interesting, and/or bizarre concurrency problems. Each has been implemented in two different programming languages (Python and Golang) and the resulting solutions evaluated and compared against each other in terms of performance (CPU space & memory used), comprehensibility (complexity of solution, lines of code), and correctness (whether the solution solves the problem, whether deadlock is possible).

In every case, a monitoring thread/goroutine was used within the program to keep track of performance metrics. Python's psutil package was used to benchmark CPU and virtual memory usage during runtime, and gopsutil, a translation of psutil into Golang, is used to in the same way. In each instance, counters are used to track the number of important operations performed during the testing period.

It is worth noting that these counters are not made thread-safe in order to avoid creating extra mutexes. It is possible for data races to occur in the incrementing of some counters, simply because the precision of their measurements is not integral to the correct operation of the program.

Classical

1 Producer-Consumer

1.1 Applications

The Producer-Consumer problem is ubiquitous in concurrent programming, appearing anywhere that data needs to be added and removed from a structure safely and synchronously. One interesting current example appears in NVIDIA GPU programming, where warp-specialized kernels can communicate and synchronize using producer-consumer named barriers in PTX assembly code in a more complex and more efficient manner than standard data-parallel threadblock-wide barriers in CUDA. [1]

1.2 Solutions & Evaluation

Two very basic finite buffer Producer-Consumer solutions are compared: one in Python, producer-consumer.py [2], and one in Golang, producer-consumer.go [3].

The Python implementation defines a single Producer and Consumer thread each, which both use a Condition object as a mutex in order to safely access the shared queue which represents the finite buffer. In both cases, buffer size was set to 10.

Since the program includes no end state by default, measuring runtime would be uninformative; instead, the program runs a monitoring thread for 10 1-second intervals, and samples CPU percentage and virtual memory size at the beginning of each interval, after which it interrupts the main thread.

The raw output of a representative execution is reproduced in python_output.txt. The first CPU percentage sample is taken before any threads are started, and the second is taken just as the threads start.

The Go implementation uses a fixed-size channel to represent the finite buffer, and single goroutines for Producer and Consumer respectively.

Similar to the Python implementation, the Go solution uses gopsutil, a translation of psutil into Golang, to tabulate CPU and virtual memory usage. It uses a monitoring goroutine to sample CPU percentage and virtual memory size at the beginning of each of 10 1-second intervals, and then exits.

The raw output of a representative execution is reproduced in go_output.txt.

1.3 Analysis

Performance

In the table below, “CPU” indicates the percentage of the CPU in use during the program’s execution, “VM” indicates the percentage of virtual memory in use, and “Productions” and “Consumptions” were incremented by 1 for each production or consumption by a thread/routine during the execution.

It is clear from the results that although the Python implementation wins out very slightly on CPU usage, the Go program outstrips it reasonably in virtual memory usage and defeats it utterly in terms of actions completed, performing over 800 times more operations due to the lightweight nature of goroutines compared to threads in Python.

In both cases, total productions outstrips total consumptions by a small margin (less than 1%), and although the Go implementation produces more unused tokens than the Python implementation, its percentage of consumed tokens is still higher owing to the massively larger volume of tokens produced overall.

Comprehensibility

In the table below, “lines” refers only to the lines of code essential for the operation of the Producer-Consumer simulation. Import/include statements, performance monitoring, and comments are not counted.

Although the Go implementation is shorter, more concise, and arguably more elegant (essential code in its produce and consume functions encompass 2 lines and 1 line respectively), it requires more specific knowledge of the Go language and its unique abstractions - specifically channels, in this case - to understand the operation and recognize that it fulfils the requirements of the problem. By contrast, the Python implementation does a better job of spelling out its execution, clearly indicating which sections are critical and what is happening at each stage. As a result, it is also easier to parse for concurrency issues.

Correctness

The Producer-Consumer problem is simple enough that ensuring correctness is a largely trivial enterprise. As long as a mutex ensures that threads have exclusive access to the buffer, the problem can be considered solved.

In the Python implementation, this is easy to confirm. Before a Producer adds to the buffer, and before a Consumer consumes, they must each acquire the shared Condition object. Neither can acquire if the other has possession, and each must release the object after performing its task.

As mentioned above, the Go implementation requires an understanding of channels before correctness can be determined. Fortunately, Go channels are thread-safe by default [4], so the "msgs" channel performs a similar type of blocking and signaling to a mutex in Python. Since the production and consumption stages are both simulated entirely by that channel, it follows that the operation is correct.

	Python	**Go**
Performance	CPU: 30.11% (2.5%) VM: 48.2% Prods: 350849 (50.1%) Cons: 350844 (49.9%)	CPU: 31.24% (2.56%) VM: 45.1% Prods: 28850450 (50%) Conss: 28850439 (50%)
Comprehensibility	39 lines Blunt but understandable	28 lines Elegant but opaque
Correctness	Correct	Correct

2 Dining Philosophers

2.1 Applications

The Dining Philosophers problem is trivialized if the two chopsticks beside each philosopher are always distributed as a pair of resources; in that case, no philosopher could pick up only half the resources needed for the eating operation. As such, the problem is particularly applicable in cases where individual processes need access to resources that cannot or should not be treated as a pair, such as in a banking transaction between two accounts where treating both as a single resource is undesirable. In this instance, the accounts are chopsticks and the transaction is represented by the philosopher, preventing two transactions from accessing the same account(s) simultaneously and overwriting balances improperly.

2.2 Solutions & Evaluation

Two implementations of the Dining Philosophers problema are compared: dining-philosophers.py [5] and dining-philosophers.go [5].

The Python implementation uses a thread to simulate each philosopher, and encapsulates the dining-thinking loop into 3 nested functions to represent thinking, attempting to dine, and dining respectively. The "all right forks" deadlock is avoided by forcing each philosopher to put down their first fork if they cannot immediately take the second, then try again for the opposite fork first, and retry until they have both forks. As a result, philosophers switch non-deterministically between left-handed and right-handed behaviour.

The raw output of a representative execution is reproduced in python_output.txt.

The Go implementation uses a channel for each fork placement around the table. Each philosopher tries to take a fork token from their adjacent channels, eats once it has two tokens, and returns a token to each channel afterwards. The deadlock is averted by flipping the channel-checking precedence for the first philosopher before beginning the simulation, making one out of the five left-handed.

The raw output of a representative execution is reproduced in go_output.txt.

Both solutions were modified to be non-terminating and induce no sleep time between states, in order to compare them more directly to each other. As above, a monitor thread/goroutine tracks CPU usage, VM usage, and total number of dining states and thinking states reached over a 10-second duration, sampling the OS status at 1-second intervals.

2.3 Analysis

Performance

The evaluation results prove similar to those of the Producer-Consumer comparison, although Python fares considerable better in executions, completing 1/40th of the Go implementation's total as opposed to 1/800th in the Producer-Consumer case. CPU usage overall was considerably higher in both cases, owing to the more complicated conditionals in the Dining Philosophers problem.

Interestingly, the Python implementation appears to have an advantage over the Go implementation in frequency of dining. This appears to indicate that the Python implementation is preferable in application settings where access to the "dining" operation is prioritized over total operations performed.

Comprehensibility

In this case, the Go program's use of channels is both creative and immediately intuitive. Tokens represent forks, channels represent place settings between philosophers, and goroutines represent philosophers, exactly as the problem dictates. Even without understanding anything about channels other than their basic purpose as pipelines to insert and remove tokens, an observer can understand the underlying metaphor. The entirety of the try-eat-think loop is accomplished in just 6 lines.

By contrast, the Python implementation falls short in its reasonable but complex solution to the deadlock problem. While it makes some sense from a high-level perspective for the philosophers to solve their problem by repeatedly trying forks and swapping which fork to try between attempts, the corresponding code is less immediately readable and the underlying intent less evident. The nested and functions help to categorize the discrete states, but naming them "run", "dine", and "dining" makes the precedence and intent difficult to immediately ascertain.

Correctness

Both instances were allowed to run for their full runtime (only several seconds) when initialized, and also for several minutes after the time constraint was removed but before the monitor thread was added to curtail execution time. In all cases, the programs ran without encountering issues or unexpected performance.

As with the Producer-Consumer problem, the Dining Philosophers problem is straightforward enough provided starvation is averted through fork precedence swaps. In both programs, these swaps are executed in different manners, but in both cases these executions appear to be thread-safe. A fork is never taken without being locked, eating is never attempted without the requisite forks, and philosophers do not communicate except by trying for fork tokens that may or may not be there. Insofar as a solution can be considered “more” correct, the Go implementation appears to allow for more consistent execution results due to its simpler deadlock fix. Trying for forks over and over again in the Python code could result in very brief instances where all philosophers have right-hand forks, and although they would then immediately release them, they could then potentially all pick up left-hand forks at once, and repeat ad infinitum. An asterisk has been added to the analysis table to represent this edge case. In practice, however, the Python program produced a higher ratio of dines to thinks, and even if a repeated switching pattern were to lead to temporary issues, the frequency of swaps indicates that the problem would eventually solve itself and is not in fact deadlock.

	Python	**Go**
Performance	CPU: 49.36% (1.3%) VM: 48.0% Dines: 677813 (50.3%) Thinks: 670606 (49.7%)	CPU: 51.43% (2.56%) VM: 45.45% Dines: 28087503 (50%) Thinks: 28098758 (50%)
Comprehensibility	64 lines Poorly-named nested functions, complicated deadlock solution	28 lines Concise, understandable channel & routine metaphors
Correctness	Correct*	Correct

Less Classical

3 Dining Savages

3.1 Applications

The unfortunately-named Dining Savages problem describes a case that occurs very frequently in real-world applications: many processes removing items from a queue while a single process adds more items in large groups. In essence, the Dining Savages problem is a particular subset of the finite buffer Producer-Consumer problem, with one bulk producer and many consumers.

One such case appears with map, reduce, or filter functions in a data streaming situation, where a large chunk of data could be made available nearly simultaneously and multiple processes could run individual operations on segments of that data. In GPU rendering, for example, a large quantity of vectors are made available all at once each timestep, and individual threads must operate on all elements as quickly as possible.

3.2 Solutions & Evaluation

Two implementations of the Dining Savages problem are compared: dining-savages.py [6] and dining-savages.go [7].

The Python implementation was created for this assignment and follows the algorithm in Allen Downey's *Little Book of Semaphores*. The cook and the savages are each assigned a thread. The cook waits for a signal from the savages that the pot is empty, then fills it with M servings of missionary and signals that it is full. The savages take servings from the pot until it is empty again, then signal to the cook to add more and wait for the cook's full signal. The implementation uses a semaphore each for the emptyPot and fullPot signals, and a mutex to control access to the number of servings in the pot.

The raw output of a representative execution is reproduced in python_output.txt.

The Go implementation once again uses channels to trivialize the algorithm: a single servings channel representing the pot is enough to encompass all needed operations. The cook adds M servings at a time to a channel of size M, while the savages remove one serving at a time until M is empty. The code is modified to allow for multiple cooks, but only one is used.

The raw output of a representative execution is reproduced in go_output.txt.

Both programs use the same initial values: a serving size of 50, a single cook, and 30 savages. Both solutions were modified to be non-terminating and induce no sleep time between states, in order to compare them more directly to each other. As above, a monitor thread/goroutine tracks CPU usage, VM usage, and total number of cooking and eating operations performed over a 10-second duration, sampling the OS status at 1-second intervals.

3.3 Analysis

Performance

Once again, the evaluation results favour the Go implementation. In this case, however, CPU usage was drastically higher for the Go program: more than twice the percentage was occupied during execution. Additionally, the Go program outperformed the Python program by a staggering amount.

Despite the apparent superiority of the Go implementation, however, these results cannot be said to be representative of a valid comparison. As discussed below, the Go program contains a logical error that impacts its correctness as a solution to the Dining Savages problem more strongly than the concurrency-related flaw in the Python program.

Comprehensibility

Like in the Dining Philosophers solution, the Go implementation here makes excellent use of channels as an intuitive metaphor for the shared pot. By contrast, the Python implementation is not nearly as opaque as in the Dining Philosophers example. The use of semaphores to signal the pot's full and empty states is apparent and easy to understand, the execution is split into functions representing each agent's rules of operation, and the actual execution steps are much simpler in the algorithm by default.

Ultimately, the Go program appears more comprehensible due only to its simplicity and one-to-one mapping of metaphor to operational elements. Although the Python implementation is slightly shorter overall, a significant portion of the Go program's length is occupied by opening and closing braces alone, making its actual readable code briefer.

Correctness

Unfortunately, due to logical errors and concurrency issues, neither program is in fact correct.

The Python program comes close but an inconsistency caught after the evaluation stage could lead to data races in certain execution patterns. While the savage appropriately locks and unlocks the mutex governing the servings left in the pot before decrementing their number, the cook does not do so before adding M servings back in. While the cook's code should only execute once the savages have taken the last serving out of the pot, it is not inconcievable that a single savage thread could be modifying the value of servings while the cook is attempting to set its value to M.

The error in the Go program is more egregious, and was left in from the original acquired solution to demonstrate the wildly inaccurate results arising from logical errors in multi-process programs. Rather than adding M *tokens* to the servings channel when empty, the cook in fact adds the *value* of M to the channel as a single token, which the very next savage thread then consumes. The cook is then signaled to do the same thing again, and the result is an implementation of the Producer-Consumer problem without the added constraint of items being added in bulk and removed one by one. This error is the reason that the Go program's evaluation results outstripped the Python program's so drastically (although it seems reasonable to posit that a correct Go implementation would also have performed more executions in the same time, just not quite so many).

	Python	**Go**
Performance	CPU: 31.69% (3.8%) VM: 48.09% Cooks: 1548 Eats: 77184	CPU: 67.78% (3.75%) VM: 44.99% Cooks: 39901752 Eats: 39873820
Comprehensibility	25 lines	27 lines
Correctness	Incorrect (missing mutex lock in cook's critical section)	Incorrect (problem requirements not met)

Not-So-Classical

4 Modus Hall

4.1 Applications

The Modus Hall problem, and similar problems such as the Baboon Crossing problem and the Unisex Bathroom problem all crop up in real-world applications anywhere that multiple processes need to access the same critical section but only certain groups can access it at a time.

4.2 Solutions & Evaluation

Two implementations of the Modus Hall problem are compared: modus-hall.py [6] and modus-hall.go [8].

The Python implementation was created for this assignment and follows the algorithm in *The Little Book of Semaphores*. Heathens and prudes are both represented by a single channel, each constantly attempting to cross through the critical section to simulate the existence of multiple agents in both parties. The code in both is symmetrical, and organized into states representing the five possible conditions at the crossing: heathens only, prudes only, two intermediate stages between them, and a neutral state. In order to cross, an agent adds itself to the queue within a mutex, checks the quantities of its own kind and the opposing kind, and updates its understanding of the program's state before crossing if it can or waiting for more of its kind if it cannot.

The raw output of a representative execution is reproduced in python_output.txt.

The Go implementation uses channels to represent each group, contained within a single struct called "path" that represents the crossing. It adds and removes members of each group to and from their respective channels as they come to the crossing. Depending on the numbers in each channel, the program decides which tokens to let through on which channel. In this way, the state of the program at any time is preserved without the need to organize execution into specific states.

The raw output of a representative execution is reproduced in go_output.txt.

Both solutions were modified to be non-terminating and induce no sleep time between states, in order to compare them more directly to each other. As above, a monitor thread/goroutine tracks CPU usage, VM usage, and total number of student crossing operations of each type performed over a 10-second duration, sampling the OS status at 1-second intervals.

4.3 Analysis

Performance

As the analysis table indicates, the Go solution once again allows for many more executions within the same time period as the Python solution. In this case, however, that amount comes at the cost of more than double the CPU load. This is due to the different ways the programs simulate streaming input in the form of students from each group. The

Python program adds a student from each group only once per iteration of that group's execution loop, but the Go program adds students from both groups simultaneously at the start of the program and then allows those students to cycle back and forth according to the balance of power.

Interestingly, the Python program maintained a more even balance between "heathens" and "prudes" over time, likely due to the fact that students were often being permitted through in single increments as the balance of power shifted fairly evenly. By contrast, the Go program permitted nearly a hundred thousand more prudes than heathens through, despite the code for each being just as symmetric. This appears to be related to the point made by its creator about the problem definition: "the situation... where the Path is occupied by increasing numbers of students happens fairly quickly." [8] In this case, further additions of one type of student will continue to shift the balance of power in that direction and continue to let that type of student through instead of the other.

Comprehensibility

The problem in this case is complex enough that both implementations required roughly a hundred lines of code. However, as in previous instances where the code length was similar, Go's implementation contains more whitespace and briefer lines, whereas the Python program is largely dense code, many conditional branches, and many wait() and signal() statements.

One benefit to the conditional branches in the Python interpretation is that they meaningfully separate the code into the finite states delineated above, which are given helpful names that contribute to the understanding of the program's structure. The addition of transitional states from one group's dominance to the other is indicative of the conditions necessary to provoke a shift in the power balance, although it necessitates the inclusion of no less than four possible unlocks for the same mutex at different stages of execution within the branches.

By contrast, the Go implementation includes no mutexes or semaphores as usual, and conditional branches are limited to switch statements based on which of the two allegiances the current student belongs to. This helps clarify the general progression through the program, and placing the critical "path" section inside a struct is useful as it clearly delineates the point of contention between the two sides.

Correctness

Both implementations appear to be correct in terms of satisfying the problem definition. Modus-hall.py diligently encases protected data within critical sections, making sure to account for all branches taken through the program, and Modus-hall.go again makes use of channels to ensure thread-safety by default. Any possible issues with expected output would appear to arise from the problem formulation itself: if the balance of power shifts too strongly towards one side, then no implementation could prevent the possibility of that side dominating completely, as it is a necessary condition of the problem. Since the side with more students is permitted - even intended - to win out, it will always be able to do so, which could very well lead to starvation on the other side. An asterisk has been added in the analysis table to reflect this potential issue. One way to fix this issue would be to add a timeout or size overload rule, where one side having control for too long or having too many students would cause the other side to "flee", allowing all students from the larger side through at once (or at least enough students to balance out the power dynamic again) before entering back into the queue to cross.

	Python	**Go**
Performance	CPU: 30.73% (0.0%) VM: 48.0% Heathens: 55492 (50.8%) Prudes: 57422 (49.2%)	CPU: 62.32% (3.8%) VM: 45.69% Heathens: 354929 (45.2%) Prudes: 430091 (54.8%)
Comprehensibility	104 lines Organized into states, but branching conditionals lack clarity	96 lines Conditionals are readable, path struct helps visualize the critical section
Correctness	Correct*	Correct*

Not Remotely Classical

5 Sushi Bar

5.1 Applications

The Sushi Bar problem exists wherever access to a resource needs to be limited and full capacity locks the resource down entirely until all processes have finished operating. A transaction involving more than two clients might share some similarities to the Sushi Bar problem: while all N clients are involved in the transaction, no other agents can access any of their details, but while some but not all of the clients are waiting on the others to arrive, they could still be engaged in other activities.

5.2 Solutions & Evaluation

Two implementations of the Modus Hall problem are compared: sushi-bar.py [6] and sushi-bar.go [9].

Sushi-bar.py follows the logic of Solution #1 from *The Little Book of Semaphores*, in which each customer thread begins eating and increments the "eating" counter if there is space, or begins waiting and increments the "waiting" counter if it cannot eat immediately. Once N threads have begun eating, a "must_wait" flag is tripped to signal all further threads to wait by default. Once a thread is done eating, it removes itself from the eating counter, and then checks to see if any others are still eating. If not, it moves up to N customers from the waiting counter to the eating counter and clears the must_wait flag iff the number of customers it has just moved is equal to N and the bar is full again.

The raw output of a representative execution is reproduced in python_output.txt.

Sushi-bar.go enforces the problem's constraints in a goroutine, and creates a struct for the restaurant containing channels for customer tokens entering and leaving the restaurant. At first, customers can enter the "in" channel until it reaches capacity, at which point its reference is set to “nil” and no further customers can enter. Once the same number of customers have left via the “out” channel, the in channel is reopened and the cycle resets.

The raw output of a representative execution is reproduced in go_output.txt.

Both solutions were modified to be non-terminating and induce no sleep time between states, in order to compare them more directly to each other. As above, a monitor thread/goroutine tracks CPU usage, VM usage, and total number of waiting and eating operations performed over a 10-second duration, sampling the OS status at 1-second intervals.

5.3 Analysis

Performance

For once, the Python implementation’s average CPU usage appears to be slightly greater overall, after factoring in the higher initial CPU load at the time the Go program was initiated. This could be an artifact of testing, considering how many more customers the Go program was able to process in the same amount of time. Multiple tests, however, indicated that the Go program had a very small but consistent advantage over the Python program in CPU usage.

Where the two implementations differ significantly is both in the customer load overall and, interestingly, in the ratio of wait cycles to eat instances in both. Sushi-bar.py allowed over ten times more eats than wait cycles, likely because the total number of cycling customers was set to 5, the same as the maximum capacity. By contrast, the number of waiting customers who got to eat after a single wait ended up precisely equal in Go. It is worth noting that this equality is not mere chance: repeated tests produced the same equality in every instance, which may be due to the architecture of the program bringing in groups of customers exactly as the same number exit, with no customer needing to enter the wait loop twice.

Comprehensibility

In this case, the Go implementation requires significantly more lines of code (even when accounting for trivial lines) due to the creation of a struct for the restaurant and corresponding functions. The meat of the implementation, however, lies once again in the channels and a single case statement to catch the in channel capacity condition. The in and out channels can be visualized as two separate doors to the restaurant, each tracking the number of customers who pass through. In this metaphor, the inward-opening door locks when all seats are full, and unlocks only when a number of customers equal to the number of seats have left via the outward-opening door.

The Python implementation is also quite simple to visualize thanks to the use of symmetrical customer code and shared variables, but it falls short in its handling of the capacity condition at a strange and unintuitive point in the process. When presenting the solution, Allen Downey points out that “the departing thread is doing work that seems, logically, to belong to the waiting threads.” [6]

Correctness

Both programs appear to be correct, and testing on each produced no unexpected results, deadlock, or starvation. In sushi-bar.py's case, all sensitive data is updated inside a mutex, and semaphores are used exclusively to pass information back and forth.

	Python	**Go**
Performance	CPU: 31.63% (1.3%) VM: 48.1% Waits: 3945 (8.4%) Eats: 43096 (91.6%)	CPU: 34.79% (6.17%) VM: 45.03% Waits: 3112211 (50%) Eats: 3112211 (50%)
Comprehensibility	38 lines Strong one-way door analogy	60 lines Confusing capacity handling method
Correctness	Correct	Correct

6 Sleep Sort

6.1 Applications

Although sleep sort itself appears to have few applications that justify its unreliability and wait time, it can be seen as something of a testament to the interesting conceptual alterations made possible by concurrent programming. The idea of performing a sort concurrently has merit in terms of applicability to map or filter functions, where the data does not necessarily need to be returned in sorted order, or has already been sorted.

Sleep sort can also help to elucidate the problems inherent in assuming that concurrent code will do exactly what it appears to regardless of the limitations or realities of the framework in which the code runs.

6.2 Solutions & Evaluation

Two implementations of sleep sort are compared: sleep-sort.py [10] and sleep-sort.go [11].

Both have been modified to create and sort a range of float values between 0 and a maximum passed via the command line, with each value along an incremental step also passed via the command line. For example, "python sleep-sort.py 10.0 0.1" and "go run sleep-sort.go 10.0 0.1" will both sort a collection of 100 numbers from 0.0 to 9.9 with increments of 0.1 between them.

Sleep-sort.py spins one thread for each number in the range, sleeps for the value of the number in seconds, and then prints out that value. Sleep-sort.go does nearly the same thing using goroutines, but first creates a waitGroup for all the routines and then exits only when all are complete, to ensure every number is printed.

A monitor thread/goroutine tracks CPU usage, VM usage, sampling the OS status at 1-second intervals during a sort of 100 floats between 1 and 10. Additionally, for reasons that will become clear in the correctness analysis below, each program was tested on sorts of floats between 0.0 and 10.0 with increments between 0.01 and 1.0, and the "sorted" result was checked for sorted order.

The raw output of a representative execution from each is reproduced in python_output.txt and go_output.txt.

6.3 Analysis

Performance

Sleep sort by definition performs in O(x+n) time, where n is the number of instructions to perform one thread's operation on one item and x is the time to sleep for the highest literal value of all items in the input.

The Go implementation outperformed the Python implementation in CPU and VM usage, despite starting from a nearly identical benchmark. However, while the Python program tended to provide a properly sorted result for items whose wait times were as little as 0.05 seconds apart, the Go implementation fell to pieces for any wait time below 1 second apart. In fact, the Go implementation would only print the results from all completed channels once every second regardless of the granularity of wait times, and the results printed could appear in any order whatsoever.

Comprehensibility

Both programs are trivial in execution, and both are nearly identical in structure. The singular difference between them in terms of comprehensibility is the addition of a waitGroup structure in the Go implementation, but the purpose of the group is not difficult to understand.

Correctness

Because in each case producing a properly sorted result hinges entirely on the amount of time given between each thread's output, there is no way to guarantee that sleep sort will produce results in sorted order given any fixed incremental wait time. The results may vary wildly according to operating system, CPU/VM load, programming environment, process scheduler, and any other variable imaginable. As a result, these implementations cannot be considered correct or relied upon in any context whatsoever. Sleep sort is an fascinating insight into the fluctuating results of seemingly predictable concurrent programs, but not a solution to the sorting problem.

	Python	**Go**
Performance	O(x + n) CPU: 5.8% (1.2%) VM: 35.35%	O(x + n) CPU: 3.48% (1.26%) VM: 31.96%
Comprehensibility	19 lines Trivial	22 lines Trivial
Correctness	Correct for wait times above 0.05 seconds apart (in general)	Correct for wait times above 1 second apart (in general)

Repository

All code used in the above solutions is available at https://github.com/jomondogu/synchronization-problems.

References

[1] Rahul Sharma, Michael Bauer, Alex Aiken. “Verification of Producer-Consumer Synchronization in GPU Programs.” http://legion.stanford.edu/pdfs/weft.pdf

[2] Producer-Consumer Problem in Python. https://www.agiliq.com/blog/2013/10/producer-consumer-problem-in-python/

[3] Go Language Patterns. http://www.golangpatterns.info/concurrency/producer-consumer

[4] Share Memory By Communicating. https://blog.golang.org/share-memory-by-communicating

[5] Dining Philosophers, Rosetta Code. https://rosettacode.org/wiki/Dining_philosophers#Python and https://rosettacode.org/wiki/Dining_philosophers#Go

[6] Allen Downey. *The Little Book of Semaphores* and accompanying sample/cleanup code. https://greenteapress.com/wp/ semaphores/

[7] Fsouza (GitHub). Dining Savages problem in Go. https://github.com/fsouza/lbos/blob/master/009-dining-savages.go

[8] Marcelo Bytes. “The Modus Hall Problem”. https://blog.ksub.org/bytes/2016/05/01/the-modus-hall-problem/

[9] Marcelo Bytes. “The Sushi Bar Problem”. https://blog.ksub.org/bytes/2016/05/08/the-sushi-bar-problem/

[10] Ssirai (GitHub). Sleep Sort by Python thread. https://gist.github.com/ssirai/4115014

[11] Arpit Bhayani (GitHub). Sleep Sort Implementation in Go. https://github.com/arpitbbhayani/go-sleep-sort/blob/master/sleepsort.go