

Ruvan Jayasinghe 012324466, Drew Stiles 010172550

April 18, 2016

CECS 327

[ruvan.jayasinghe@student.csulb.edu](mailto:ruvan.jayasinghe@student.csulb.edu) [drew.stiles@student.csulb.edu](mailto:drew.stiles@student.csulb.edu)

Starvation-Free Dining Philosophers

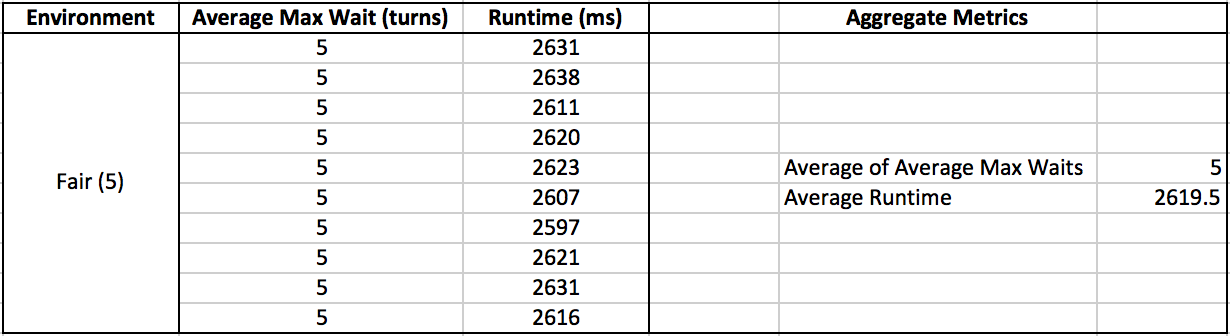
**Summary:**

A solution to the dining philosophers problem, a classic concurrency problem in Computer Science, was presented to us for modification. In the unmodified version of the solution, philosophers that wish to eat were subject to starvation: when one philosopher finished eating and signaled the next philosopher to eat it would do so with no regard to how long (in seconds or turns elapsed) since that philosopher had last eaten. In our implementation, we have attempted to use turns elapsed since a philosopher has last eaten in order to determine which philosopher to signal.

We utilize a global counter that is incremented every time a philosopher eats. We then record the value of that counter to an array where indices correspond to each philosopher and update it every time the philosopher eats again. When each philosopher attempts to take the sticks to eat, it will ensure that its priority is higher than both of its neighbors before testing to see if it can eat and then ultimately eating or waiting. If the philosopher is made to wait, it will attempt to regain the lock and take the sticks again. When the philosopher has finished eating and puts the sticks down, it will then signal its neighbor should that neighbor be capable of eating at the moment. This signal may or may not be dropped as it depends on whether or not the signaled philosopher has sent the *await* message to that condition object. With this framework in place, the remainder of this paper will explore a starvation-free implementation for accessing the critical section of this classic problem.

**Experiment 1: (5 philosophers, 20 turns)**

For this portion of the project the runtime environment contained 5 philosophers each executing 20 turns. Each turn guarantees a philosopher will eat, although the ordering of the eat assignments are dependent upon the *takeSticks* implementation used for the program execution. Below is a table outlining the relative performance of the fair *takeSticks* implementation involving 5 philosophers. In summary, the relative execution durations for both implementations were largely equivalent in their average over the 10 trials used to collect the samples for this experiment.

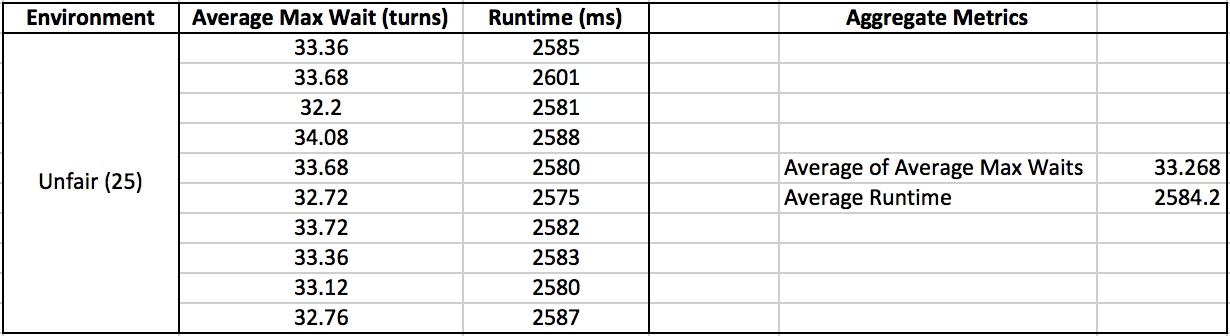


*Collected metrics for the fair takeSticks implementation involving 5 philosophers*

Though not shown the unfair *takeSticks* implementation performed only slightly better in its runtime, registering an average runtime of 2573.7 ms approximately 59 ms faster in the average case. However, the fair *implementation*  performed better in terms of its fairness and predictability for the anticipated maximum wait for a potential philosopher. As can be seen in the table above, the maximum wait was never over 5 for any execution of the fair *takeSticks* implementation. In the unfair implementation average max waits varied between 6.0 and 6.6 turns. This trend of lower waits and longer runtimes will begin to manifest itself more prevalentlyin the subsequent experiments where the runtimes for the unfair implementation will experience constant rate growth as compared to the linear growth of the fair implementation runtimes. However in contrast, the fair implementation will always provide a upper limit for exactly how long a thread will wait to eat, equal exactly to number of philosophers in the environment.

**Experiment 2: (25 philosophers, 20 turns)**

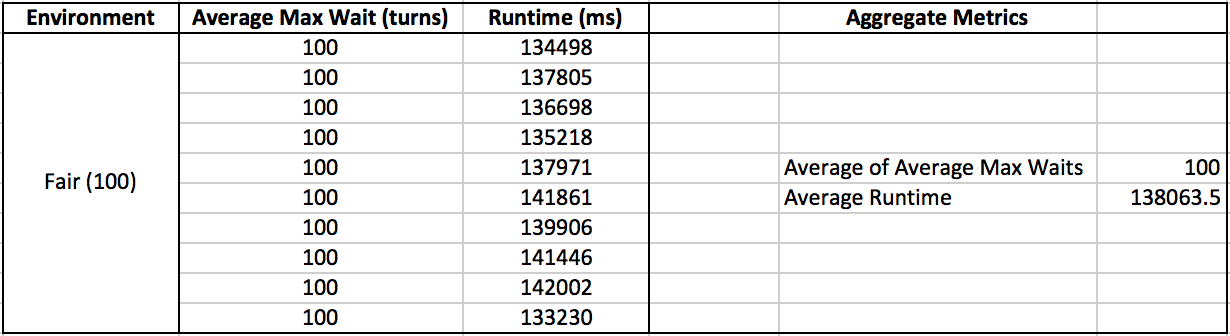
Moving the number of executing philosopher to 25, while maintaining the same number of turns for each philosopher, began to expose the differences between the two implementations of *takeSticks*. Most notably was the relatively unchanging execution duration for the unfair implementation. That is, although the number of philosophers increased by a factor of 5, the program completed execution in approximately the same amount of time. This is likely due to the fact that although the environment became more populated with the increase in philosophers, the effect upon which philosopher would be eating had no relation to whether that philosopher should be eating. Concretely, any philosopher who could eat at the time they wanted to eat would therefore eat, reducing the under-utilization of the critical section. It is this mechanism which contributes to both the small execution time, along with the starvation potential for the unfair *takeSticks* implementation. As can be seen in the table below, when allowing anyone who can eat, to eat, there is potential that some philosopher ready to eat must wait a disproportionate number of turns before they are signaled. This is likely caused by the influence of locality in assigning the sticks to waiting philosophers. If there exist a set of philosopher who repeatedly release and acquire the sticks more consistently due to lack of an assignment policy, then those who continuously fail to acquire the lock will experience starvation in the form of large number of turns between their eat turns, which in the fairest case would see a max wait of exactly 25 since the longest a philosopher is guaranteed to wait is exactly equal to the number of philosophers eating.



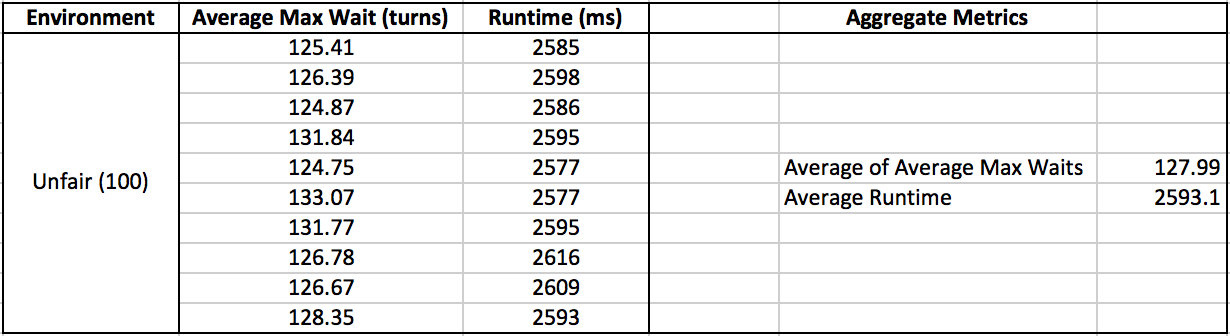
*Collected metrics for the unfair takeSticks implementation involving 25 philosophers*

**Experiment 3: (100 philosophers, 20 turns)**

Following the trends of the two previous experiments the strengths and weaknesses of each implementation are highlighted most here. More precisely the fair *takeSticks*  implementation reveals its strength as a strictly fair algorithm (FCFS critical section access). In providing this fairness however, the execution times have grown linearly with the number of executing philosophers as is displayed in the concluding graphs below. In comparison, the unfair *takeSticks* implementation has not deviated far from its execution duration of ≈2600 ms while the average max wait times among all philosophers now vary between ≈25-35 turns longer than that that of the fair implementation. With these three experiments in place, we can begin to forecast how exactly these two implementations fare in terms of the two primary metrics under consideration in this report. As it appears, the cost of guaranteed fairness and predictability is a large performance hit in terms of the programs relative throughput. That is, it will take far longer to pipe N philosophers through their respective turns if fairness is required. Also at the cost of fairness is the large resulting number of idle (busy-wait/polling) philosophers who continuously loop until the state of the system is such that they may eat. This can be observed in the code below, but at the high level this is due to the need for a philosopher to constrain itself when its neighbors appear hungrier than itself. Prohibiting this progression is ultimately the reason we see lineargrowth in the program runtimes for the fair implementation. Specifically, as in the 100 philosopher case here, philosophers must eat in such a manner that requires all other philosophers to eat before one philosopher may eat another time, adding a degree of rigidity that serves as a large bottleneck in the overall system’s progression.



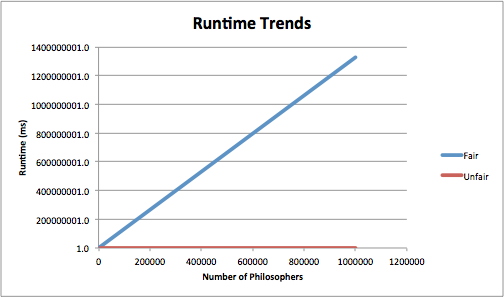
*Collected metrics for the fair takeSticks implementation involving 100 philosophers*



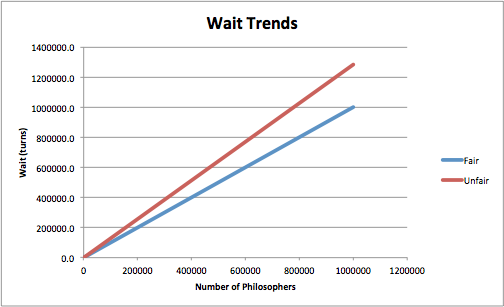
*Collected metrics for the unfair takeSticks implementation involving 100 philosophers*

**Conclusion:**

To sum up the above results, the cost of fairness in the suggested implementation here is a significant bottleneck resulting in much longer program runtimes. Alternatively, the unfair implementation here presents greater starvation risk as the environment becomes more populated. While further optimizations were not explored in this general solution, intuitively it seems as though the trending growth in wait times may be subject to some improvement, however the fact of coordination and FCFS eat assignments means that any fair implementation will likely experience some lower bound resulting from the waits required to guarantee linearization of the eating philosophers. The diagrams below highlight the trends for each of the implementations in terms of waits and runtimes, extrapolating these values out to a million philosophers in the environment. As can be seen the decision in which algorithm is applicable within a given context will largely be determined by the demands of the environment as both implementations have exactly opposite strengths and weaknesses.



*Fair implementation runtime grows linearly compared to constant unfair implementation growth*



*Unfair implementation experiences larger rate of change in average max wait compared to ideal (fair)*

**Source:**

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.concurrent.atomic.AtomicInteger;

**import** java.util.concurrent.locks.\*;

**public** **class** Philosopher **extends** Thread {

**public** **static** **int** *WAITING* = 0, *EATING* = 1, *THINKING* = 2;

**public** **static** **final** **int** ***TURNS*** = 20;

**public** Philosopher

(Lock l, Condition p[], **int** st[], **int** num,

**int** ID, AtomicInteger cnt, **int**[] apps, **double**[] max) {

// default implementation

lock = l;

phil = p;

states = st;

NUM\_PHILS = num;

id = ID;

// starvation free implementation

**this**.counter = cnt; // reference to timestamp allocator

**this**.appetites = apps; // reference to all others appetites

**this**.max = max; // shared array for writing this philosophers max wait

waits = **new** ArrayList<Integer>(); // reference to this philosophers waits

appetites[id] = lastAte = counter.incrementAndGet(); // assign timestamp

}

**public** **void** run() {

**for** (**int** k = 0; k < ***TURNS***; k++) {

**try** { Thread.*sleep*(100); }

**catch** (Exception ex) {}

takeSticksFairly(id);

**try** { Thread.*sleep*(20); }

**catch** (Exception ex) {}

putSticks(id);

}

// write max wait for this philosopher

max[id] = Collections.*max*(waits);

}

**public** **void** takeSticks(**int** id) {

lock.lock();

states[id] = *WAITING*;

**try** {

**if** (!canEat(**this**.id)) {

phil[id].await();

}

**else** {

// go eat

}

eat();

} **catch** (InterruptedException e) {

System.*exit*(-1);

} **finally** {

lock.unlock();

}

}

**public** **void** takeSticksFairly(**int** id) {

**while** (**true**) {

states[id] = *WAITING*;

**boolean** hungrierThanRight =

(appetites[rightof(**this**.id)] >= appetites[**this**.id]);

**boolean** hungrierThanLeft =

(appetites[leftof(**this**.id)] >= appetites[**this**.id]);

**if** (hungrierThanLeft && hungrierThanRight) {

lock.lock();

**try** {

**if** (canEat(**this**.id)) {

// go eat

}

**else** {

phil[**this**.id].await();

}

eat();

**break**; // from while

} **catch** (InterruptedException e) {

System.*exit*(-1);

} **finally** {

lock.unlock();

}

}

**else** {

**continue**; // trying to eat

}

}

}

**private** **void** putSticks(**int** id) {

lock.lock();

**try** {

states[id] = *THINKING*;

**if** (states[leftof(id)] == *WAITING*

&& states[leftof(leftof(id))] != *EATING*) {

phil[leftof(id)].signal();

}

**if** (states[rightof(id)] == *WAITING*

&& states[rightof(rightof(id))] != *EATING*) {

phil[rightof(id)].signal();

}

} **finally** {

lock.unlock();

}

}

**private** **boolean** canEat(**int** id) {

**return** (states[id] == *WAITING*

&& states[leftof(id)] != *EATING*

&& states[rightof(id)] != *EATING*);

}

**private** **void** eat() {

System.***out***.printf("Philosopher %3d is eating\n", id);

**int** count = counter.incrementAndGet();

waits.add(count - lastAte);

lastAte = count;

appetites[id] = lastAte;

states[id] = *EATING*;

}

**private** **int** leftof(**int** id) { // clockwise

**int** retval = id - 1;

**if** (retval < 0) // not valid id

retval = NUM\_PHILS - 1;

**return** retval;

}

**private** **int** rightof(**int** id) {

**int** retval = id + 1;

**if** (retval == NUM\_PHILS) // not valid id

retval = 0;

**return** retval;

}

**private** Lock lock;

**private** Condition phil[];

**private** **int** states[];

**private** **int**[] appetites;

**private** **int** NUM\_PHILS;

**private** **int** id;

**private** AtomicInteger counter;

**private** **int** lastAte;

**private** ArrayList<Integer> waits;

**private** **double**[] max;

} // end class Philosopher

**import** java.util.concurrent.atomic.AtomicInteger;

**import** java.util.concurrent.locks.\*;

**public** **class** PhilTest {

**public** **static** **int** *WAITING* = 0, *EATING* = 1, *THINKING* = 2;

**public** **static** **final** **int** ***NUM\_PHILS*** = 5;

**public** **static** **void** main(String a[]) {

*init*();

**long** start = System.*currentTimeMillis*();

*execute*();

**long** end = System.*currentTimeMillis*();

System.***out***.printf("\nMax Wait (Avg): %.2f turns\nRuntime: %d ms", *getAverage*(*max*), (end - start));

}

**public** **static** **void** init() {

// construct private data structures

*phil* = **new** Condition[***NUM\_PHILS***];

*states* = **new** **int**[***NUM\_PHILS***];

*appetites* = **new** **int**[***NUM\_PHILS***];

*max* = **new** **double**[***NUM\_PHILS***];

// initialize environment threads

**for** (**int** k = 0; k < ***NUM\_PHILS***; k++) {

*phil*[k] = *lock*.newCondition();

*states*[k] = *THINKING*;

}

}

**private** **static** **void** execute() {

Philosopher p[] = **new** Philosopher[***NUM\_PHILS***];

// begin

**for** (**int** k = 0; k < p.length; k++) {

p[k] = **new** Philosopher(*lock*, *phil*, *states*,

***NUM\_PHILS***, k, *counter*, *appetites*, *max*);

p[k].start();

}

// synchronize

**for** (Thread t : p) {

**try** { t.join(); }

**catch** (InterruptedException ex) { ex.printStackTrace(); }

}

}

**private** **static** **double** getAverage(**double**[] arr) {

**double** avg = 0.0;

**for** (**int** i = 0; i < arr.length; i++) {

avg += arr[i];

}

**return** avg / arr.length;

}

**private** **static** Lock *lock* = **new** ReentrantLock();

**private** **static** Condition[] *phil*;

**private** **static** **int**[] *states*;

**private** **static** AtomicInteger *counter* = **new** AtomicInteger(0);

**private** **static** **int**[] *appetites*;

**private** **static** **double**[] *max*;

} // end class PhilTest