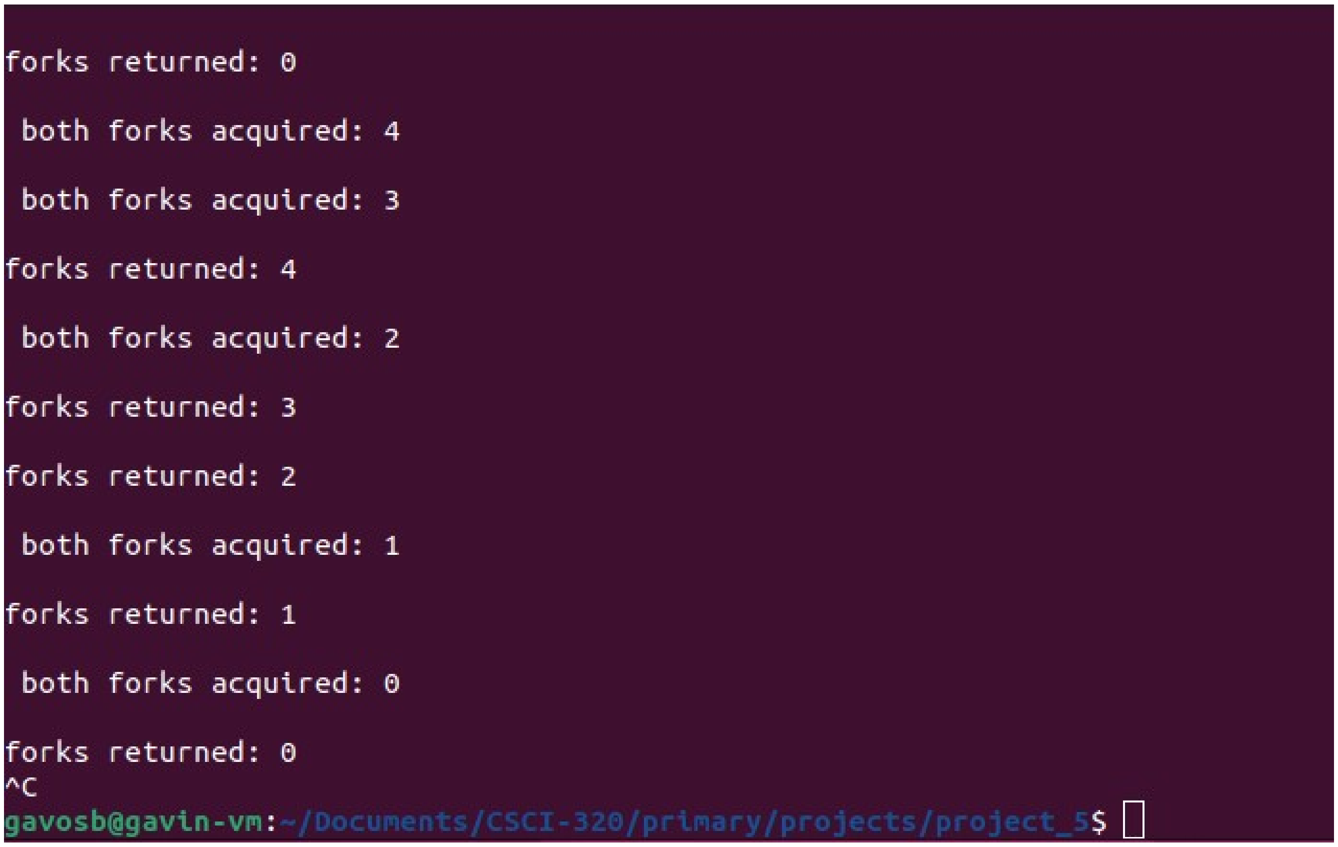
Dining Philosophers Algorithm 1

This ultimately results in deadlock as soon as all philosophers wait on their left fork at the same time. In this example, as soon as both forks are returned, they're grabbing their left fork.

The program basically deadlocks immediately when philosophers total less than 5. Behavior doesn't seem to change when even or odd.



```
forks returned: 0

 both forks acquired: 4

 both forks acquired: 3

forks returned: 4

 both forks acquired: 2

forks returned: 3

forks returned: 2

 both forks acquired: 1

forks returned: 1

 both forks acquired: 0

forks returned: 0
^C
gavosb@gavin-vm:~/Documents/CSCI-320/primary/projects/project_5$
```

Dining Philosophers Algorithm 2

The semaphores in this implementation are redundant, as everything will be handled by the bitmap in the first place, which serves as a bottleneck. Starvation can be shown to occur; this arises from the fact that only different patterns of philosophers can eat at any given time. For example, if there are 5 philosophers, there are only two able to eat at the same time. This means that a pattern can develop where 4 philosophers alternate between eating, leaving one out, though still being able to eat if prerequisite patterns run before it. Such being the case, the starvation has a chance to occur, but will not last long; Thus, in every case of running, one process will have slightly fewer accesses of the shared resources by the end of the process.

Consider this input. The count is 32, sum 439806, the mean 13744 and variance 36732 from a standard deviation of 192.

Philosopher	**Servings**	**Z-Score**
0	13813	0.36
1	13708	-0.188

2	13665	-0.412
3	13826	0.428
4	13856	0.585
5	13686	-0.302
6	13509	-1.226
7	13741	-0.015
8	14000	1.336
9	13808	0.334
10	13829	0.444
11	13797	0.277
12	13833	0.465
13	13917	0.903
14	13789	0.235
15	13664	-0.417
16	13977	1.216
17	14045	1.571
18	13889	0.757
19	13966	1.159
20	13611	-0.694
21	13599	-0.756
22	13642	-0.532
23	13128	-3.214
24	13272	-2.462
25	13779	0.183
26	13675	-0.36
27	13570	-0.908
28	13741	-0.015
29	13835	0.475
30	13802	0.303
31	13834	0.47

Anything above 1.65 can be considered overfed and below -1.65 starved. Philosophers 23 and 24 evidently starved quite a bit. This was in all likelihood not due to randomness, but because of a fundamental flaw in the algorithm.

Of course, this definition relies on starvation being more so like malnutrition. The processes are still able to run at times, but can get repeatedly locked out of the dominant pattern.

This also highlights a flaw in my analysis. At first glance, it just appears that the algorithm works perfectly fine. However, because of this statistical analysis, I can only imagine that there must be some sort of discrete and deterministic process from the initial threads that can cause one or two threads to only run when a given set of conditions have already been established; i.e., some combination of threads must grab their resources before the starved thread can grab theirs. I am sort of at a loss for how exactly this happens.

There are no distinguishable differences from running the program with 0.5 sleep times for 60 seconds, 0.5 sleep times for 20 seconds, and 0.2 sleep times for 60 seconds. The algorithm does not appear to behave differently with even or odd philosopher counts.

Dining Philosophers Algorithm 3

This algorithm does not cause deadlock nor starvation. Each philosopher takes turns sitting at the table, distributing eating time. Because there are n+1 forks available, there is always at least one philosopher that is able to eat since if no one is waiting on the last fork while all others are waiting on another fork, it will mean that they will have acquired both forks.

The algorithm is less efficient than the official solution to the problem. They have to wait on the table in order to eat in the first place, whereas the original solution just has one philosopher upsetting the process by picking up forks in a different order. This can potentially mean drastic performance costs if eating is a lengthy process while thinking less so; kind of like how some dogs go about their lives (like mine).

The algorithm does not appear to behave differently with even or odd philosopher counts, nor with different sleep times.

Overview of Dining Philosopher Problem

The best solution is doubtlessly the official one, but of the algorithms here, #3 seems to work best for fairness. #2 still works, but has a bottleneck around the bitmap for every eat which also holds back anyone else trying to eat (this is not the case for the one “gate” being around the table itself). It also seems to suffer from starvation due to a pattern determined from the start. #1 simply doesn’t work due to deadlocking.

The Shuttle Problem

My implementation allows for the shuttle to drop off passengers at a second bus stop, and then loop back to the beginning. To do this, I made the shuttle a set of shared resources accessed in the same scope of mutual exclusion, and did the same for each bus stop. Several auxiliary resources are used as overhead, denoted in the comments of the program.

Improvement could be made by having the attendees wait around at each area rather than immediately waiting on the bus stop, resulting in passengers at B immediately getting back on the bus. However, this technically achieves the specifications of the assignment.

I was asked to see if I could drop either the passenger mutex or the waiting counter mutexes, but I have found both to be necessary save for drastically modifying the program (and introducing possible errors). The reason for this is also marked within the relevant code section.

I don’t believe any processes are starved. They wait at the bus stops, thus being woken up with equal chance. The program does not deadlock. There are some bottlenecks, like how only one attendee boards the bus at a time due to all the mutexes involved.