**OSP programming assignment 1:**

**S3540567 – Timothy Tran**

**Github: https://github.com/spacediscotqtt/s3540567-OSP-assignment1**

**Notes, Issues, and limitations:**

Both for problem A and E, the Valgrind input indicates that all blocks are freed on exit as indicated here:  
Text

Description automatically generated  
figure (1): Valgrind output for A

Text

Description automatically generated  
figure(2): Valgrind output for E

As pointed out in the code, solution for Problem A was based on possible solution provided by lecturers and referenced from [https://shivammitra.com/c/producer-consumer-problem-in-c/#](https://shivammitra.com/c/producer-consumer-problem-in-c/). For problem A, Semaphores were used over pthread conditional signaling to achieve process synchronization and avoid busy-waiting in terms of checking the size of the buffer in order to indicate whether a process should be sleeping or not to signal the use of a thread. The use of semaphores provides a simple elegant solution to the sleeping condition of a producer consumer problem of which indicates whether there are empty spots in the buffer/bucket to fill and to sleep if there aren’t for the producer, and whether there a filled slot in the buffer/bucket and to sleep if there aren’t for the consumer.

In the Smoker-Agent problem there may be potential busy waiting solutions that could have been implemented, however I am unclear about what constitutes such issues in the context of my work at this time. The implementation of “making it fair” may have been a potential busy waiting issue.

**Producer and consumer**

The problem dictates several conditions

-an array of size 10 which is a bucket/buffer that holds items

-items are produced by the producer, In this case, it’s a random number between 1-100

- 5 producers produce these numbers, and 5 consumers consume these numbers. Which are printed on the screen

-5 producers run concurrently as do the 5 consumers.

-if the bucket is full, the producers will sleep as there are no spaces to produce items, if the bucket is empty, the consumers will sleep as there are no items to consume.

-the program exits after 10 seconds

In my solution for the producer consumer problem, semaphores are key to ensuring that no CPU memory is wasted regarding producing resources that would be discarded. As seen here in the producer method:

Text

Description automatically generated

The semaphore empty Slots dictates whether the producer should be asleep based on whether there are empty slots that are available to be filled, by default there are ten empty slots and corresponding semaphore full Slots. If there are no empty slots (which is to say the slots are incremented up to the size of the bucket) then the bucket is full, and the producers would sleep. The mutex is then locked so the producers don’t immediately compete for resources and follow a prioritized order of producing products. After the production, the semaphore fullSlots are incremented, which indicates that the previously empty spot has been filled nonspecifically. In this scenario, the sleep has been implemented to track the time of the program to ensure it terminates after ten seconds.

The bucket array is synonymous with a buffer in the original specs for producer consumer, items that are “produced” by the producer is put into the buffer or which is marked as parallel by the slots of the empty and full semaphores.

A screenshot of a computer

Description automatically generated with medium confidence

Conversely, the consumer method follows a very similar mode of processor for the producer albeit in reverse, of which the semaphore fullSlot indicates the thread should sleep if there are no fullslots e.g resources to consume. the thread is then locked if the conditions are met where a individual consumer is able to consume a item from the buffer and increment the emptySlot semaphore to indicate that something was consumed.

**problem: starvation and deadlock**

starvation:

There are conditions implemented to ensure that the producer-consumer problem is starvation free:

-a consumer will sleep until there is full slots to consume

-a producer will sleep when there are no empty slots to fill

-the producer will always produce when there are empty slots

-the consumer will always consume if there are full slots to consume

-both consumer and producer threads run concurrently

The starvation problem is never allowed to occur, in an implemented producer-consumer, resources that prevent execution are intrinsically linked to each other and of equal priority. With semaphores an empty bucker would force the consumer to sleep, but the producer would continue to process items to the bucket, wherein a full bucket would cause the producer to sleep, leading the consumer process consuming items from the bucket. Therefore these 2 methods deny the ability for starvation as the resources are produced and consumed in tandem to each other.

Deadlock:

A deadlock is solved in the solution, as the semaphores implemented both imply opposites conditions for sleep, as such if one process activates sleep for a thread, for example a full bucket, the consumer would still process Which in turn would wake up the corresponding thread to activate and vice versa. Often, both consumers and producers are active, and where not possible always 1 thread of the process can act. Due to the contradictory and dependent nature of the sleeping conditions for both consumer and producer, both cannot sleep at the same time and thus it cannot cause a deadlock.

**Real life applications for producer-consumer solutions**:

The applications for the solutions of the producer-consumer are a common one in terms of multi-process synchronization and resource management, an example of it being used in a similar context would be that between a network device and an operating system. The network device would act as a producer, producing data at certain rates and places it in the buffer (which originally is called bucket in the solution), wherein the operating system acts a consumer and consumes the data from the same buffer.

A more macro example would be main information server that produces singular requests to client computers which would be seen in IT’s phone helplines or automated temp agencies. The server would receive or “produce” tasks that are consumed by the client computers and are therefore “consumed” however the bucket in this case would be an unspecified if not unlimited length and unlike the solutions the rate of production and consumption for each task would be much more variable.

**Smoker and Agent**

The problem dictates several conditions:

-3 smokers with an abundance of 1 ingredient each, being: paper, tobacco and matches respectively

-an agent who contributes 2 random ingredients on a table.

-smokers are only able to grab ingredients from the table

-smokers are only able to smoke if they possess all 3 ingredients for a cigarette

-a smoker who grabs the wrong corresponding ingredients cannot smoke

-an agent cannot replenish the table unless a smoker has smoked

thus, if the wrong ingredients were taken by the wrong smoker, both the smoker and agents would be unable to proceed, leading to a deadlock.

A solution to the problem proposed is that the only smoker who has the required missing 3rd ingredient would be allowed to smoke. This code was built off a proposed pseudo code as seen here: http://www.cs.umd.edu/~hollings/cs412/s96/synch/smokers.html. This is done by calling the smokers only after the ingredients have been placed by the agent of which smokers then respond by checking the table concurrently. Implementation of this method would therefore work in rounds. The order of which a single round places out is as shown:

1. Text

Description automatically generated  
the agent seizes the table if it is empty and places 1 of possible 3 combinations onto the table. The combinations are represented as integers. And that selection is added to an array of a history of the past 3 table placements

2.

Text

Description automatically generated

After the table is filled, a cond signal named fullTableCond is called indicating that the table is filled and to wait until the table is empty as indicated by the emptyTableCond. Thus the agent thread will sleep until the conditions are called such that the table is empty.

3.

Text

Description automatically generated

The smokers remain asleep if they somehow reach the table faster, and continues to wait until the table is filled, in which case the smoking method iterates through the smokers. Smokers who don’t have the required ingredient are told to sleep and wait for a new round whereas the smoker who matches the table number consumes the ingredients. The conditions are reset, and a new round can commence.

4.

Text

Description automatically generated

To make it fair, a history of the previously smokers was collated, for every 3 rounds, each smoker will be able to smoke atleast once, thus the table generation is semi-random.

**Starvation and deadlock:**

Starvation isn’t allowed to occur; the processes of the cigarette smoker solution are dependent of each other where no process can recursively call itself leading to a starvation scenario.

The original issue of deadlock in the proposed cigarette smoker problem is where a smoker is able to grab ingredients that do not match it’s own, leading to the agent and the smoker unable to proceed forward. By allowing only the correct smoker to process itself, is the deadlock avoided as only the correct smoker can smoke thus a deadlock cannot occur.

**Practical application:**

The solution is multifaceted in it’s practical application as there have been high profile problems to due with immature seizure of resources. In the case of the heartbleed issue, where a server machine acts as the agent and the client acting as smokers, with the heartbeat requests being agent requests. A unwanted side effect of the cigarette smoker issue is a deadlock, wherein the unintended issue of the heartbleed issue is an integer overflow. Both lead to unwanted outcomes in terms that there have been no verification of the table contents and in the case of the server, the memory requests.

A real-life example of the algorithm is present in amazon fulfillment centers, the delegation of certain items takes special equipment provided by fulfillment agents who are in this case cigarette smokers. If the sorters, acting as agents, freely provide the items unsorted, agents that do not have the required tools to fulfill the order may lead to freezes in productivity equivalent to a deadlock. The fulfillment agents would be incentivized to fulfill as many orders as possible leading them to grab, if unsorted/unchecked and order that requires a tool/ingredient they do not have. Although in this scenario, the delegation team would continue delegating tasks instead of sleeping as I the case with the cigarette smoker problem.

Sample outputs for A and E:

Text

Description automatically generated

Text

Description automatically generated