CSOPESY S13 **06/14/22**
Group 2 **Sir Arren Antioquia**
LOPEZ, Angel L.
JADIE, Joshue Salvador A.

Project 2 - Synchronization Problems Documentation

Synchronization Technique

To synchronize the execution of the threads and coordinate between blue and green threads, both **mutex locks** and **counting semaphores** were implemented. Using **Python** as the language to implement the solution, the **`threading`** library was utilized. From the library, we used the **`Lock`** object which is equivalent to a mutex lock**,** and both the **`Semaphore`** and **`BoundedSemaphore`** objects which are equivalent to counting semaphores.

Synchronization Variables

Below are the variables that we have created for implementing synchronization along with their corresponding usages:

Variables	Usage
`n`	The **number of slots** in the fitting room. Used for initializing `slots` which is a **counting semaphore** (**`BoundedSemaphore` object**).
`b`	The **number of blue threads** to be executed. Used to initialize and execute blue threads as well as to check if all blue threads have been completed already.
`g`	The **number of green threads** to be executed. Used to initialize and execute green threads as well as to check if all green threads have been completed already.
`blue_mutex`	A **`Lock`** object (mutex lock). To allow only **blue** threads to access the fitting room.
`green_mutex`	A **`Lock`** object (mutex lock). To allow only **green** threads to access the fitting room.
`slots`	A **`BoundedSemaphore`** object (counting semaphore). To allow **`n`** threads to access the fitting room at a time given that there are **`n`** slots.

`limit`	**Integer** variable that defines the **number of accesses** each thread color has before passing the fitting room access to the other color. This is done to **avoid starvation**.
`blue_limit`	A **`Semaphore`** object (counting semaphore). To allow `limit` (value) **blue** threads to access the fitting room before passing the access to the green threads.
`green_limit`	A **`Semaphore`** object (counting semaphore). To allow `limit` (value) **green** threads to access the fitting room before passing the access to the blue threads.
`blue_executed`	**Integer** variable that indicates the current number of completed **blue** threads.
`green_executed`	**Integer** variable that indicates the current number of completed **green** threads.

Constraints and Solutions

[A] ***There are only*** **n** ***slots inside the fitting room of a department store. Thus, there can only be at most*** **n** ***persons inside the fitting room at a time.***

```
# counting semaphore, which also indicates that there are n fitting room slots only
# bounded so that its current value will not exceed its initial value (n)
slots = BoundedSemaphore(n)
```

To satisfy **constraint A**, a **counting semaphore** was implemented which will allow **n** threads to enter the fitting room with **n** slots. From the threading library of Python, a **`BoundedSemaphore`** object called **`slots`** was initialized with a value of **n**. This object was used instead of the **`Semaphore`** object to ensure that its current value will **not exceed** its initial value which is **n**. It will then strictly ensure that only at most **n** threads can enter the fitting room at a time.

```
# get a slot in the fitting room, wait if no slot is available yet
slots.acquire()

# print blue thread's information
print(current_thread().name)

# simulate work in critical section
sleep(1)

# blue thread is done with its work, increase number of blue threads completed
global blue_executed
blue_executed += 1

# release slot
slots.release()
```

When executing the blue threads, for example, a blue thread will first try to acquire a slot in the fitting room by calling the **`acquire()`** function of **`slots`** before doing work in its critical section. After acquiring a slot, it will then do work in its critical section and **increment** the number of blue threads **completed** since it has already finished its work. If there are no slots available yet then it will wait until a blue thread inside the fitting room is done with its work and has released its slot by calling the **`release()`** function of **`slots`**.

[B & D]

There cannot be a mix of blue and green in the fitting room at the same time. Thus, there can only be at most n blue threads or at most n green threads inside the fitting room at a time.

The solution should not result in starvation. For example, blue threads cannot forever be blocked from entering the fitting room if there are green threads lining up to enter as well.

```
# to restrict access of fitting room to blue threads only
blue_mutex = Lock()

# to restrict access of fitting room to green threads only
green_mutex = Lock()

# fitting room access limit per color, to avoid starvation (change value if needed)
limit = n + 5

# counting semaphores for fitting room access limit
# it is designated that blue threads will access the fitting room first (if both b and g are > 0)
# if there are no blue threads, execute green threads only
if b <= 0:
    green_limit = Semaphore(limit)
    blue_limit = Semaphore(0)
# else, blue threads will go first, having an initial value of limit and 0 for green threads
else:
    blue_limit = Semaphore(limit)
    green_limit = Semaphore(0)
```

As observed in the code snippet above, **two mutex locks** and **two counting semaphores** were implemented. An **integer** variable named **`limit`** was also created which defines the number of blue/green threads allowed to enter the fitting room before passing the access lock to the other color. Both **`blue_mutex`** and **`green_mutex`** are **`Lock`** (mutex lock) objects used to restrict fitting room access to only one color. Meanwhile, both **`blue_limit`** and **`green_limit`** are **`Semaphore`** (counting semaphore) objects used to allow **`limit`** blue/green threads only in accessing the fitting room.

If both **`b`** and **`g`** are **`> 0`**, **`blue_limit`** will be initialized with a value of **`limit`** and **`green_limit`** with a value of **`0`**. These indicate that **`limit`** **blue** threads will always access the fitting room first per run. After **`limit`** (or possibly all if **`b < limit)`** blue threads have been completed, the **`last`** blue thread will notify (signal) **`limit`** (or all if **`g < limit`**) green threads afterward so that they can also access the fitting room after the **lock** has been **released**. Since the initial value of **`green_limit`** is **zero** (**`0`**), the **`Semaphore`** object was used instead of **`BoundedSemaphore`** so that its current number can exceed its initial value.

In the case that there are no blue threads (**`b == 0`**) in the input then **`green_limit`** will be initialized with a value of **`limit`** instead of **`blue_limit`**. This will execute the green threads immediately or else the program will get **stuck** because there are **no** blue threads that will **signal** the green threads.

`execute_blue()`

```python
# get access lock to fitting room, decreasing number of accesses
# for this color
blue_limit.acquire()

# if green threads have claimed the fitting room, wait until
# lock is released
while green_mutex.locked():
    pass

# if fitting is currently not claimed by both blue and green threads
if blue_mutex.locked() == False and green_mutex.locked() == False:
    # let blue threads claim the fitting room
    blue_mutex.acquire()

    # blue thread is the first to enter the fitting room as having acquired
    # the lock
    print("Blue only.")
```

`execute_green()`

```python
# get access lock to fitting room, decrease number of accesses
# for this color
green_limit.acquire()

# if blue threads have claimed the fitting room, wait until
# lock is released
while blue_mutex.locked():
    pass

# if fitting room is currently not claimed by both green and blue threads
if green_mutex.locked() == False and blue_mutex.locked() == False:
    # let green threads claim the fitting room
    green_mutex.acquire()

    # green thread is the first to enter the fitting room as having acquired
    # the lock
    print("Green only.")
```

In our code implementation, we have **two (2)** functions: **`execute_blue()`** and **`execute_green()`**. Both functions have the same algorithm, the only difference is that **blue** threads are executed in `execute_blue()` while **green** threads are executed in `execute_green()`. With the initial values set for `blue_limit` and `green_limit`, **`limit`** **blue** threads will be able to go first and acquire the lock with **`blue_mutex`**, restricting the access to blue threads only. All of the **green** threads will have to wait after calling the **`acquire()`** function of **`green_limit`** until the **`last`** blue thread in the set is done with its work.

`execute_blue()`

```python
# get updated number of executed green threads
global green_executed

# if blue thread is the last in the fitting room, give access lock to green threads
if blue_executed == b or (blue_executed % limit == 0 and green_executed != g):
    # fitting room is currently empty
    print("Empty fitting room.")

    # if there are remaining green threads to be executed, give access lock to green thread
    if green_executed != g:
        # signal "limit" green threads so that they can access the fitting room
        for i in range(limit):
            green_limit.release()

        # release access lock
        blue_mutex.release()
# else if all green threads have already been executed, no more access limit for blue threa
elif green_executed == g:
    blue_limit.release()
```

`execute_green()`

```python
# get updated number of executed blue threads
global blue_executed

# if green thread is the last in the fitting room, give access lock to blue threads
if green_executed == g or (green_executed % limit == 0 and blue_executed != b):
    # fitting room is currently empty
    print("Empty fitting room.")

    # if there are remaining blue threads to be executed, give access lock to blue threads
    if blue_executed != b:
        # signal "limit" blue threads so that they can access the fitting room
        for i in range(limit):
            blue_limit.release()

        # release access lock
        green_mutex.release()
# else if all blue threads have already been executed, no more access limit for green thre
elif blue_executed == b:
    green_limit.release()
```

In execute_blue(), for example, after the **last** blue thread has entered the fitting room and finished its work, it will then check if all **green** threads have already been completed by checking the value of **`green_executed`**. If all green threads are not yet completed, the last blue thread will signal **`limit`** green threads by calling the **`release()`** function of **`green_limit`** and then release the access lock afterward by calling the **`release()`** function of **`blue_mutex`**. Else, all blue threads will be continuously executed until they are all completed. This is also done for **green** threads in **`execute_green()`** if the same case happens. Additionally, the two mutex locks were also implemented so that **`limit`** green threads will access the fitting room together and not one by one after the **last** blue thread signals **`limit`** green threads, and vice versa.

This solution satisfies **constraint B** as the two mutex locks restrict the fitting room access to blue threads and green threads alternately. It also satisfies **constraint D** at the same time since only **`limit`** blue threads will be executed at first then **`limit`** green threads are executed immediately afterward, and vice versa. The solution ensures that if for example there are **ten-thousand (10000)** blue threads and **five (5)** green threads, the green threads **do not have to wait** until all **10000** blue threads are completed. **`limit`** blue threads will be completed first then the 5 green threads will be executed immediately before completing the rest of the blue threads.

[C] ***The solution should not result in deadlock.***

Considering the **four (4)** conditions needed to be met in order for a deadlock to occur, all the conditions are met except for **hold and wait** and **circular wait**.

Based on the solutions for **constraints A, B,** and **D**, the **hold and wait** condition was not satisfied since the green threads are **not holding** any resources that the blue threads need while waiting for **`limit`** blue threads to finish their work. The same happens when it is the other way around.

For **circular wait**, it was also not satisfied since the way the solution works is that it will first start with the blue threads acquiring the lock and accessing the fitting room slots. The green

threads will then wait for `limit` blue threads to finish their work and release the lock without holding anything. The same will be done by `limit` green threads and the access lock will be given back to the blue threads again, with the blue threads also not holding any resources that the green threads need during their execution. This is done repeatedly until all threads are completed. The execution between blue and green threads is being done **alternately** and **no resources are being held** by a color while waiting for the other color to pass the access lock.

With these two conditions **not satisfied**, a **deadlock** will **never occur** in the implemented solution.