Write Up

Blocking

The code starts with declaring the counting semaphore structure; under that, we have a thread condition variable, mutex variable, and semaphore value. After that, declaring the sauce bowl semaphore variable and structure philo will contain the philosopher's data. In the main program, we have created threads based on the number of inputs. Then calling the function philosopher() from pthread_create and passing it addresses an integer variable, which refers to the philosopher number. The philosopher function will first convert the number passed as a void * into an integer.
Then it will be called the sem_wait function, which first checks if the resource is available. If it is available, the resource is allocated to the philosopher by decrementing the value of semaphores. If all the resources are allocated, i.e., the semaphore value is <0, then the thread will wait for the sem_signal. If the semaphore's value is changed to <0, indicating that the semaphore is blocked.
Then, we are allowing the philosophers to eat.
Finally, we are freeing the semaphores by calling the sem_signal function so that the other threads that are waiting can use the resources. If it returns a positive value, the semaphore is unlocked successfully. After the philosopher is done, it will join the threads back to the main process.

Non-Blocking

In this program, the same thing will happen, but instead of waiting, it will return and try again to get the resources. In the sem_wait function, if the semaphore value becomes <0, it will continue trying to get the resources until the sem_signal function makes the semaphore value >0. After that, it will try to acquire a lock using try lock mutex, which will return 0 if a lock

on the mutex object referenced by mutex is acquired, then it will come out of the lock, decrementing the semaphore value. Similarly, in the sem_signal function, it will first try to acquire a lock, then increment the value and unlock the lock.