For the implementation of the semaphores, I have used the pthread library as instructed in the question. For both the variants, the semaphores have attributes called, my_semaphore mutex, value which is the value of the semaphore, int wakeups which counts the number of pending signals. The wakeup symbolises the number of threads that have been woken up but not resumed its execution. The changes for the implementation are described below.

Blocking variant:
In the blocking, the thread hangs until it has some kind of answer. It keeps waiting until the resource is available. And thus, the process continues. The system calls can thus take quite long than usual.

In the make_semaphore function, the semaphore's attributes are initialised.
In the sem_wait function, the mutex is locked, the lock is taken by the thread, and the thread value is decremented. Then it sees if the value is less than zero, the process is suspended as no more process can then enter into the critical section. Then the mutex is unlocked.

In the sem_signal, when the mutex is locked, the value of the semaphore is incremented by one and if the value is non-negative, a process is selected and woken up. The queue is not required in any of the function because conditional pthread is used to so that the thread executes signal.

Non-blocking variant:

In this variant, the program does not wait for the I/O to complete. Instead it terminates the process whenever such condition arrives. "The completion of the I/O is later communicated to the application either through the setting of some variable in the application or through the triggering of a signal or call-back routine that is executed outside the linear control flow of the application." (read from http://faculty.salina.k-state.edu/tim/ossg/Device/blocking.html)

To define the mutex lock, I have used trylock, it is similar to lock except that if the mutex object referenced by mutex is currently locked (by any thread, including the current thread), the call shall return immediately.

The make_semaphore function is same as the blocking variant.

In sem_wait function, the semaphore is locked first, and then if the try lock executes, the lock is unlocked, and the semaphore is then incremented. It then checks if the number of threads that have been woken up is less than one, so that the more threads can be woken up.

In sem_post function, the semaphore, there is no thread to give the signal, as that has been already handled by the try lock, therefore we do not require any condition and just lock the mutex, there is no requirement for decrementing as the try lock ensures this, that the race condtion does not occur.

For the whole implementation, I have understood and referred to the TheLittleBookOfSemaphores. In dining philosophers problem, k is taken as input and the deadlock condition is removed by giving the odd philosophers first the left fork and then right fork, and the even philosophers first the right fork and then the left fork, for bowls there are two counting semaphores.