Operating Systems
Project 1: Sirens and Pirates

Frederic-Gerald Morcos – 4-1805 – E10
Christine Maher – 4-1083 – E10
Nardine Basta – 4-626 – E10

- **NOTES:**
  - More information is available as code comments (for more details).
  - To run the program, pass a number (a multiple of 4) as a command-line argument (ex: java Main 8).
  - Please compile it with JCreator and run it using command prompt. Use JDK 1.5.

- **The Synchronization problem:**
  - Java uses monitors for process synchronization**.** A built-in mechanism ensures that only one Java thread can execute an object's synchronized method at a time (i.e. mutual exclusion). The mechanism also allows threads to wait for resources to become available, and allows a thread that makes a resource available to notify other threads that are waiting for the resource**.** If a thread arrives at the beginning of a monitor region that is protected by a monitor already owned by another thread, the newly arrived thread must wait in the entry set. When the current owner exits the monitor, the newly arrived thread must compete with any other threads also waiting in the entry set. Only one thread will win the competition and acquire the monitor.
  - Our problem here is that we wanted to ensure that the threads are mutually excluded and that they don't stand in each other's way. We used synchronized methods for that purpose trying to use Java's features of synchronization and mutual exclusion.

- **The main Classes we implemented:**
  - **Semaphore:**
    - This class implements the semaphore, with methods down() and up() as they are known. Used to solve the synchronization problem introduced by the pirates and sirens.

  - **Barrier:**
    - This class implements the barrier, with a reset() method that will “restart” it and reset it's variables, an arrive() method that will be called when a process wants to send a signal that it reached the barrier and a destroy() method that will cause the barrier to be released (will wake up all of the blocked threads).

  - **Process:**
    - Implements the pirates and sirens, extends Thread so we get the java

    threading capabilities. The run() method will run the thread, that will let it enter the boat (after doing a down() on a mutex for mutual exclusion in using the boat).

  - **Main:**
    - This is where all of the program starts. A legal() method that will use checkForPirate() and checkForSirens() to check if a pirate or siren can enter the boat, respectively. A rowBoat() method that will be used by the last process entering the boat so it start rowing and finally a main() method that will initialize the processes depending on user input (as arguments to the program) and start them.

- **Semaphore Implementation:**
  - A semaphore is an integer that has a maximum and minimum value, in our case, we do not care about the maximum value as the number of processes is unknown. As for the minimum value, it is zero.
    - The down() method will keep checking if the semaphore is equal to zero, if so, will keep sending sleep signals to it's calling threads, otherwise will decrement the semaphore.
    - The up() method will increment the semaphore and notify any of the sleeping threads.

- **Barrier Implementation:**
  - A barrier is usually used for processes to set a limit point so they cannot advance with other processes lagging behind. In other words, for processes to wait for each other.
    - The reset() method will reset the barrier's variable a list of blocked_threads.
    - The arrive() method is called by a process to signal that it reached the barrier. Also, lets the barrier take note that it should wake it up when released.
    - The destroy barrier releases the barrier and wakes up all of the blocked threads that have been waiting for it.

- **How we used semaphores and barriers to solve the sirens and pirates problem:**
  - **Semaphores:** we used the semaphores to protect the critical region which is entering the barrier (boat). All the processes after being created fight to enter the boat (the barrier implicitly), we put here a semaphore mutex to prevent more than one thread from entering at the same time. Each thread does a down() on the mutex before entering the boat if it's allowed to enter (i.e. If the number of threads on the boat are less than 4) it updates the counts and then releases the mutex. If there are already 4 passengers on the boat, it pauses for a while until the threads in the boat are released and then is notified to be able to enter the boat
    NOTE: The thread here pauses while having the mutex which means that any other thread that will try to down() the mutex to enter the boat will sleep on the mutex.

    - **Barriers:** the barrier here is used to gather the requested number of threads so the boat is allowed to row. If the number of threads (passengers) required is not completed yet, the threads that have already arrived will sleep inside the barrier until the number of threads is complete. At that time all of the waiting threads are now allowed to traverse the barrier. Here it is used to prevent the boat from rowing unless it has 4 threads on it.

- **Sources we used to complete this project:**
    - www.javadocs.org
    - www.koders.com
- The program runs fine, only a little glitch is missing, the part where processes should run randomly. In other words, they part where, each execution of the program should give a different result. In the order we start the threads, they are executed, we tried to solve this in several ways (by adding an initial barrier then releasing it when all of the processes are started) but wouldn't give any results. Though no matter what order of pirates and sirens is given to the program, it runs fine and ends gracefully :)