REAL TIME SYSTEMS FOR AUTOMATION

PARTY ANIMALS

Assignment 1

Problem Analysis

Identifying waiting conditions is the first thing to do to start problem analysis since each of these conditions will roughly correspond to one semaphore in the implementation. We can clearly see that at least two waiting conditions exist (which can also be seen as synchronization points between threads). First, each person attending the toast has to wait for the other four people before shouting "Skol". Second, no one can start drinking until all the other four people of the group have shouted "Skol". However, there is another important condition to meet which is the fact that only one toast can occur at a time in order to limit neighborhood disturbance. This implies that when a toast starts all other people should wait until the toasting group has finished.

Design

Implementation uses one semaphore for each waiting condition/synchronization point to properly synchronize threads plus a couple of mutexes to protect critical sections.
We will start by addressing the first two conditions we detailed above since they are simpler and represent the core of the assignment. Before diving into the actual implementation, it is worth noting that each condition can be implemented within its own subroutine due to the fact that it is independent from the others. Indeed, there is no way for a thread that is waiting for shouting to unblock one waiting for drinking and vice versa. In addition, the logic behind the implementation of both subroutines is almost identical. They start by acquiring a mutex lock in order to update the value of a counter. This counter is used to keep track of how many threads have already reached the synchronization point so it is immediately checked to see if it is still less than 5 (the toasting group size). If this is the case, we have to wait for more people to join us for toasting. So, the executing thread releases the lock and performs a wait operation on the semaphore linked to this condition and then it is blocked. The order of these two operations is critical, otherwise no other thread would be able to acquire the lock anymore, causing a deadlock. On the other hand, if the counter is equal to

five, the thread represents the last person needed to start toasting and there is no need for it to block. The thread decrements the counter (in order to keep it consistent) and leaves the critical section by releasing the lock. Finally, we have to increment the value of the semaphore to allow any other waiting thread to proceed. A previously waiting thread could then complete synchronization by posting on the semaphore to unblock other waiting threads. Clearly, this should happen only if there are threads still waiting. Thus, the executing thread has to acquire the mutex lock and check the counter before posting on the semaphore. Given this, the first two conditions described in the problem analysis are fulfilled. The third condition is a bit more subtle since it is not independent from the other two, but it influences them instead. In fact, satisfying the third condition means that we have to block any thread approaching a toast if one has already been started or it is currently held. Then, once the toast finishes, we have to wake those blocked threads. The implementation of these two operations makes use of another semaphore in order to properly synchronize threads. Every thread starts by issuing a wait operation on the semaphore. The semaphore has to be initialized to one instead of zero, unlike the others, otherwise no thread would be able to proceed. The first thread passing through the semaphore will then post to it before blocking on the first condition semaphore to unblock the next thread. The unblocked thread will do the same and the process will go on until five threads have been unblocked. This could be seen as an important limitation at first, but it is not, since the first thing that threads will do after going through the semaphore is entering the counter critical section. However, if another thread has already entered the critical section, threads approaching the critical section will be blocked on mutex acquisition until the thread that is currently holding the lock releases it. As we have seen, from a concurrency perspective in practice nothing changes. From this point on, we continue as in the first condition implementation so everything proceeds as described in that section until the toast finishes. Lastly, the last thread leaving the second condition procedure posts on the third condition semaphore unblocking a new thread and as such starting a new toast.

Statistics

Statistics are implemented as thread local variables and returned on thread exit as argument to `pthread_exit`. In particular, during initialization each thread allocates memory on the heap for a struct type which holds the number of drinks consumed by that person together with its name. This data has to live on the heap otherwise the pointer

referring to it becomes dangling as soon as the thread exits. At the end of the simulation, the main thread collects the statistics of each thread, sorts them according to the number of drinks and outputs the result.