Peterson Lock in Java

In the proof above we assumed all variables are atomic registers, the read and write operations take effect at a single point in time between the invocation of an operation and the operation completing. The Java memory model is different. We first provide an intuitive explanation of why the above proof of mutual exclusion still holds under the Java memory model. Further below you will find a more formal explanation. Simplified we can say that:

1. Program order is observed within a thread.
2. A read of an atomic or volatile variable synchronizes with all "previous" writes to it.
3. A read of an atomic variable always sees the "last" value written to it.

Thus, since we read victim (a volatile) before we read flag, we will also see the changes to flag, and all the ordering and visibility guarantees we used in the first proof are still valid.

However, starvation freedom no longer holds if flag is not volatile. Lets say thread B is in the critical section and A is stuck in the while loop. B will eventually exit the critical section and write $flag[B] = false$, but now, since flag is not volatile and there are no happens-before edges between the two threads, we have no guarantee that thread A will ever see that write. The Java memory model allows for an execution where A never sees the write and starves.

More formal explanation (optional)

As hinted in the shorter explanation, we do not need to reinvent the proof, but just have to justify the key arguments using the Java memory model. The first step is to express the assumption $W_A(victim = A) \rightarrow W_B(victim = B)$. In the Java memory model, every execution has a "synchronization order" (SO), which totally orders all the synchronization actions that were performed in the execution. SO was less of a focus in the lecture, but it is an important part of the memory model and needed for this proof. We now assume

$$W_A(victim = A) \overset{SO}{\rightarrow} W_B(victim = B),$$

i.e. we assume that B's write comes after A's in SO. Because the two writes are synchronization actions and SO is a total order, either this or the other analogous case must occur. From this we can derive the following:

- $R_B(victim)$ must return B, since the read of a volatile variable must see the last write to it *in synchronization order* (see 17.4.7, point 5 in the specification). (Note that happens-before is not enough to deduce this. Both writes to victim happen-before the read, but are otherwise not ordered in happens-before.)

- $W_A(victim = A) \overset{SW}{\to} R_B(victim)$, since a write to a volatile variable synchronizes-with all subsequent writes to the variable *in synchronization order* (see 17.4.4, bullet point number two in the specification). Then

  $$W_A(flag[A] = true) \overset{HB(PO)}{\to} W_A(victim = A) \overset{HB(SW)}{\to} R_B(victim) \overset{HB(PO)}{\to} R_B(flag[A])$$

  and by transitivity $W_A(flag[A] = true) \overset{HB}{\to} R_B(flag[A])$, guaranteeing us that indeed thread B must see A's flag as set. To be precise, we can conclude this because all other writes to $flag[A]$ happen-before the write $W_A(flag[A] = true)$ (due to program order) and "happens-before consistency" (you can search for this in the specification) tells us that a read may only see the last write(s) in happens-before order (or any other write not in happens-before, but there is no such write here).

This then again gives us our desired contradiction, namely that thread B cannot have read $false$ in the while loop.