Mutty Open Questions

Lock1:

a) ¿Qué situación de bloqueo no deseado puede ocurrir con esta implementación (lock1) cuando hay alta contención?

El problema principal de **lock1** es que no tiene un mecanismo para evitar **interbloqueos (deadlocks)**. Cuando hay muchos procesos compitiendo por el acceso a la sección crítica (alta contención), es posible que se genere una situación de espera circular, lo que provoca un interbloqueo (**deadlock**): si dos procesos esperan mutuamente la respuesta del otro, provocando una parada indefinida ya que ninguno procede.

b) Indique dos situaciones que puedan llevar a la retirada de un proceso (es decir, que un proceso se canse de esperar para acceder a la sección crítica). Suponga que los procesos no fallan

Situación 1: **Deadlock**:

! Un proceso no recibe la confirmación de bloqueo en 8000 ms.

En lock1, cuando se da lugar a un Deadlock un proceso puede **abandonar la espera** si no recibe el permiso para entrar en la sección crítica dentro del tiempo de espera definido (**8000 ms en esta implementación**). Cada proceso está esperando respuestas de los demás, pero nadie responde porque están esperando respuestas también. El temporizador de cada proceso llega a 8 segundos, por lo que se retiran uno por uno.

Situación 2: **Contención continua**

! Un proceso con un tiempo de trabajo elevado monopoliza la sección crítica durante este tiempo.

Un proceso se mantiene en la zona crítica **durante un tiempo superior al tiempo de espera máximo de otro proceso.** En este caso el proceso en espera se retira de la espera (waiting) después de que se exceda el tiempo máximo de espera.

Como veremos más adelante, esto también puede ocurrir con lock 2 y lock 3.

1. P1 envía solicitudes a P2 y P3.
2. P2 responde rápidamente, pero la respuesta de P3 se retrasa debido a congestión en la red.
3. P1 sigue esperando, pero el tiempo de espera (?withdrawal) se agota antes de que llegue la respuesta de P3.
4. P1 decide retirarse, aunque en teoría podría haber obtenido el bloqueo si la red hubiera sido más rápida.

Lock2: c) Justifique cómo su código (lock2) garantiza que solo un worker esté en la sección crítica en cualquier momento, especialmente en la situación complicada descrita antes.

mirar sol-ex-parcial-17.pdf pregunta 6 apartado c

Lock2 garantiza la exclusión mutua **dando prioridad a los ID inferiores (tienen mayor prioridad)**. Cuando un proceso en estado 'wait' recibe una solicitud con un ID inferior:

1. El proceso actual cede ({ok, Ref}).

2. Vuelve a solicitar [**reenvía su propia solicitud** al nodo prioritario (From ! {request, … } reiniciando su espera], asegurando que sólo un proceso (el de mayor prioridad) acumule todas la confirmaciones (respuestas *ok*) necesarias para entrar a la sección crítica (evita que 2 procesos estén en la región crítica al mismo tiempo). mirar sol-ex-parcial-20.pdf pregunta 2 apartado c)

3. Una vez que el proceso entra a la zona crítica (en estado **held,** referencia en la función del código) , las nuevas solicitudes se posponen (guardadas en la lista Waiting) y los otros procesos no podrán acceder a la zona crítica hasta que el proceso libere el candado (release), ya que no habrán recibido todos los *ok* para hacerlo.

Así, aunque múltiples procesos compitan, solo uno (el de mayor prioridad) obtiene el candado en cada ciclo, garantizando exclusión mutua. Como esta versión trabaja con prioridades, se podría decir que no respeta el **happened-before order**, pero si el **safeness (**safety**)**.

d) ¿Las situaciones descritas en la pregunta anterior b) aún llevan a una retirada del proceso en esta implementación (lock2)?

Sí, aunque lock2 evita interbloqueos (**deadlocks**) mediante prioridades, los workers aún pueden retirarse por:

1. **Timeout:** Si un proceso de baja prioridad (ID alto) es desplazado repetidamente por IDs menores, puede no recibir el candado (release) antes de los 8000 ms.

2. **Demoras acumuladas:** En alta contención, incluso procesos prioritarios podrían enfrentar demoras en recibir confirmaciones, superando el timeout.

En resumen, debido a que habrá procesos más prioritarios que otros, por lo que los procesos poco prioritarios pueden sufrir frecuentemente situaciones en las cuales no podrán acceder a la zona crítica en mucho tiempo y su tiempo máximo de espera expirará.

e) ¿Cuál es el principal inconveniente de esta implementación (lock2)?

Esta implementación no tiene en cuenta el orden temporal en el que se pide el lock: La **inanición (starvation)** de procesos con IDs altos. Al priorizar IDs menores de forma absoluta, procesos con IDs mayores pueden quedar en espera (perpetuamente si el sleep time es muy pequeño) si hay solicitudes constantes de IDs menores.

Provoca un gran desbalance de tiempo que ocupa cada instància en la zona crítica: esto viola la equidad (**fairness**) provocando que algunas instancias no consigan entrar en la zona crítica o que el tiempo que tarden los nodos menos prioritarios a acceder sea muy elevado. Ejemplo:

Se puede dar la situación en que haya **3 procesos P_1, P_2, P_2, donde $ID_{P1} < ID_{P2} < ID_{P3}$**, y tanto **$P_1$ como P_2** están en estado **open** (sin pedir el lock), y **P_3 inicialmente pida el lock**. A esto, **P_1 y P_2 le responden OK**, pero alguno de los **OK tarda en llegar**, por lo que a **P_3 no se le concede el lock**. Después, resulta que **P_1** también **pide el lock**, por lo que este deberá ser el que lo reciba ($ID_{P1} < ID_{P3}$). En este caso, **P_3** tendrá que enviar el **OK a P3 y volver** a enviar su request, para que se quede en la **waiting list de P_1** y **espere a que P_3** acepte darle el lock.

Lock3:

f) Justifique si esta implementación de candado (lock3) requiere enviar un mensaje de solicitud adicional (además del mensaje ok) cuando un proceso en estado de espera (wait) recibe un mensaje de solicitud con el mismo tiempo lógico de otro proceso con mayor prioridad.

```
wait(Nodes, Master, Refs, Waiting, TakeRef, MyId, MyReqTime, MyClock) →
    receive
        {request, From, Ref, ReqId, ReqTime} →
            MyNewClock = max(MyClock, ReqTime), % Actualiza el reloj local

            %% CLAVE DEL ALGORITMO DE LAMPORT: Ordenamiento por timestamps
            %% Si la solicitud tiene un timestamp menor O igual pero con ID menor (mayor prioridad)
            if ( ReqTime < MyReqTime ) or (( ReqTime == MyReqTime ) and ( ReqId < MyId )) →
                From ! {ok, Ref}, % Cede el paso a la solicitud con prioridad
                wait(Nodes, Master, Refs, Waiting, TakeRef, MyId, MyReqTime, MyNewClock);

            true →
                %% Deja la solicitud en espera si tiene menor prioridad
                wait(Nodes, Master, Refs, [{From, Ref}|Waiting], TakeRef, MyId, MyReqTime, MyNewClock)
            end;
```

No hará falta enviar un mensaje de solicitud adicional, porque con esta implementación se tiene en cuenta el tiempo lógico de envío de la request (ReqTime [logic timestamp]) para enviar el OK. Cabe recordar que, según las reglas de Lamport, si dos procesos tienen el mismo timestamp, se desempata por el ID numérico (ID menor tiene más prioridad).

No se envía una nueva solicitud por parte del proceso receptor; a diferencia de lock 2 donde se envía solicitud tras ceder el paso, aquí el receptor **continúa esperando** las confirmaciones (ok) pendientes de la solicitud original. Es decir, la prioridad se resuelve localmente sin reiniciar el proceso de solicitud.

Por lo tanto, no puede ocurrir que el proceso “perdedor” haya recibido un OK previo al de ese momento, por parte del proceso con ID menor que acaba de ganar el desempate.

g) ¿Las situaciones descritas en la pregunta b) aún llevan a una retirada del proceso en esta implementación (lock3)?

Los deadlocks **no** pueden suceder ya que en caso de empate en tiempo se usa la metodología del lock 2 basada en prioridades para hacer el desempate. Sin embargo, aunque se ordenan las solicitudes por timestamp lógico e ID, el tiempo de 8000 ms sigue aplicándose por tanto la segunda situación puede seguir sucediendo: un proceso ocupando la zona crítica durante más tiempo que el tiempo de espera de un proceso que quiere acceder.

h) Los workers no están involucrados en el reloj de Lamport. Según esto, ¿sería posible que a un worker se le conceda acceso a una sección crítica antes que a otro worker que emitió una solicitud a su instancia

de candado causalmente antes (suponiendo orden happened-before)? (Nota: los workers pueden enviar mensajes entre ellos independientemente del protocolo de exclusión mutua)

Si mirais las respuestas del github, estas dicen que no es posible. Si preguntáis Deep Seek o Chat GPT, dependiendo de cómo lo habéis entrenado os dirán que sí es posible.

! Si mirais la solución del control parcial 26 Abril 2023 (sol-ex-parcial-23.pdf) pregunta 2 apartado b) os dirá que si es posible, lo que pasa es que la pregunta no tiene tanto detalle como esta. La cosa es que aqui pone “suponiendo orden happened-before”.

```
%% Estado "abierto": el lock está disponible y puede ser solicitado
%% MyClock mantiene el valor del reloj lógico local
open(Nodes, MyId, MyClock) ->
    receive
        {take, Master, Ref} ->
            MyNewClock = MyClock + 1, % IMPORTANTE: Incrementa el reloj al solicitar el lock
            Refs = requests(Nodes, MyId, MyNewClock), % Envía solicitudes con timestamp
            wait(Nodes, Master, Refs, [], Ref, MyId, MyNewClock, MyNewClock);
        {request, From, Ref, _, ReqTime} ->
            %% IMPORTANTE: Actualiza el reloj local al máximo entre el actual y el recibido
            MyNewClock = max(MyClock, ReqTime),
            From ! {ok, Ref}, % Responde inmediatamente en estado abierto
            open(Nodes, MyId, MyNewClock);
        stop ->
```

Cuando un worker envía take, el lock incrementa su reloj local (MyClock + 1) y lo usa en las solicitudes.
Sin embargo, las interacciones externas (e.g., mensaje de w1 a w2) no actualizan MyClock, dejando el reloj desincronizado del contexto causal.

Sí es posible. Los workers no participan en el reloj de Lamport, por lo que eventos externos al protocolo de lock no actualizan los timestamps. Para evitarlo, se requeriría que todas las interacciones entre workers actualizarán los relojes del lock (para “sincronizarse entre sí”), algo que no ocurre en la implementación actual.

Ejemplo:

Trabajador w1 entra en la sección crítica y envía un mensaje de toma a su instancia de bloqueo l1, notificando también a w2. Al recibir el mensaje, w2 decide entrar en la sección crítica y envía un mensaje take a l2.

Dado que estos mensajes *take* no incluyen el reloj lógico, no se puede determinar el orden “happened-before”.
El acceso se decide según el tiempo lógico de cada instancia de bloqueo al recibir el mensaje *take* y enviar los request.

Si el tiempo de l2 es menor que el de l1 (o si son iguales y el ID de w2 es menor), w2 accederá primero, aunque la solicitud de w1 haya ocurrido antes.