ON THE SET OF MOLECULES OF NUMERICAL AND PUISEUX MONOIDS

MARLY GOTTI AND MARCOS M. TIRADOR

1. Barriers en C#

Permite que múltiples tareas trabajen de manera cooperativa en un algoritmo en paralelo a través de múltiples fases. Ninguna hebra puede comenzar una nueva fase del algoritmo hasta que las restantes hebras no hayan alcanzado la barrera (todas hallan completado la fase anterior).

El constructor más simple de la clase toma por argumento un entero, que representará el máximo número de participantes (hilos) que pueden estar esperando por la barrera en cada fase del algoritmo. Los métodos `AddParticipant()` y `RemoveParticipant()` permiten aumentar en una unidad la cantidad la cantidad máxima de participantes.

El método fundamental de la clase es `SignalAdndWait()`. Este es invocado por una hebra cuando llega a un punto en que está lista para moverse a la siguiente fase. Este advierte a la instancia de *Barrier* de que un nuevo participante ha llegado a la barrera. Cuando hayan llegado una cantidad de participantes igual al número máximo entonces la barrera entra en la post-phase. Aquí es donde incrementa el número de la fase, y le da la señal a todos los participantes de que pueden entrar en la nueva fase. Adicionalmente mediante el otro constructor de Barrier se puede establecer una acción que se ejecutará en cada post-phase.

El último método relevante que posee es el `Dispose()` que permite liberar los recursos usados por la barrera cuando esta no se necesite usar más. Si se intenta usar la barrera después de que esta haya sido "disposed", lanzará una excepción del tipo *ObjectDisposedException*. Los otros tipos de excepciones lanzadas por los métodos de la clase son *InvalidOperationException* y *BarrierPostPhaseException*. La primera de estas ocurre cuando en la barrera están esperando más participantes que el máximo permitido o cuando se invoca alguno de sus métodos estando en la post-phase. La segunda ocurre cuando la acción establecida a ejecutar entre fases lanza una excepción. Esta es capturada y envuelta (wrapped) en una excepción del tipo mencionada y relanzada en cada uno de los hilos (participantes).

Date: December 11, 2022.

2010 *Mathematics Subject Classification.* Primary: 20M13; Secondary: 06F05, 20M14.

Key words and phrases. Puiseux monoid, numerical monoid, atomic monoid, atomicity, factorization, molecule, atom, BFM, FFM, UFM.

Para implementar la clase Barrier se hizo uso de la clase `Monitor`. Para ello en el cuerpo del método `SignalAndWait()` se usa `Monitor.Wait` para hacer que la hebra que lo ejecuta espere a que el resto llegue a la barrera. Cada vez que llega una hebra nueva se incrementa la cantidad de participantes que han llegado. Cuando se alcanza el máximo lo primero que se ejecuta la acción prefijada (si alguna), desde la última hebra en llegar. Entonces se incrementa el número de la fase y se invoca `Monitor.PulseAll` para que todas las hebras reanuden su ejecución. Sin embargo, se tuvo mucha sutileza para manejar la mayor cantidad de casos posibles en que pudieran ocurrir errores de sincronización, como podría ser la muerte por inanición. Se maneja cada situación de la manera más correcta posible según criterio de los autores, y se lanza la excepción adecuada en cada caso, según lo descrito sobre estas anteriormente.

2. Problema de los 5 Filósofos

El problema presentado en esta sección es un problema clásico para ejemplificar los problemas de sincronización, el cual fue tomado de ([**?**, Epígrafe 17.7.2.1]).

Descripción: Cinco filósofos se pasan la vida comiendo y pensando, sentados en una mesa circular con un tazón común en el centro que contiene "infinitos" espaguetis y con un tenedor colocado entre cada par de filósofos. Cada filósofo alterna a su antojo el comer y el pensar, sin comunicarse con los demás. Para comer, cada filósofo necesita de los dos tenedores (el que tiene a su izquierda y el que tiene a su derecha) por lo que puede ocurrir que tome uno de los dos tenedores y tenga que esperar por el otro que está ocupado por su vecino. Al terminar de comer el filósofo debe colocar de vuelta los tenedores sobre la mesa. Puede ocurrir una situación en que cada filoósofo tenga ocupado el tenedor a su izquierda y al querer tomar el tenedor a su derecha este esté ocupado por el filósofo a su derecha quien también está en la misma situación y, así, de modo circular. Entonces todos están bloqueados sin poder comer por falta de un tenedor.

Problema: Diseñar un método con el cual los filósofos puedan alternar entre comer y pensar eternamente, sin previo conocimiento de cuándo los otros filósofos desean comer y, claro, sin tener que aumentar la cantidad de tenedores porque es obvio que si cada uno tiene dos tenedores no habría problema.

2.1. Solución en C#.

En [**?**, Epígrafe 17.7.2.2] podemos encontrar una solución parcial al problema. La misma consiste en usar un mediador (mesero) que determine en cada momento quién puede o no usar los tenedores según los soliciten. De esta forma se elimina la posibilidad de interbloqueo y por tanto que el programa quede esperando infinitamente. Sin embarago, se dice que es una solución parcial ya que no elimina la posibilidad de muerte por inanición. La propuesta dada consiste en que solo un filósofo pueda pedirle al mesero ambos tenedores, en cada momento. Sin embargo la decisión de qué filósofo es atendido primero queda en manos del sistema operativo, y

no sabemos como se comporta este, pudiendo darse el caso en que deje a uno de los filósofos esperando sin ser atendido nunca, o relativamente pocas veces.

Para evitar ese problema en este trabajo se propuso contar con una cola en la que los filósofos se van colocando cuando necesitan comer. La idea es que cuando un filósofo decida comer este revise la cola a ver si no hay nadie que quiera alguno de sus dos mismos tenedores (y que tendría por ende prioridad), así como revisa a quienes están comiendo para ver que no estén ya ocupados. En caso de que coger los tenedores sea consistente con lo anterior, este los toma y se sienta a comer. Si no, se coloca al final de la línea. Por supuesto, mientras un filósofo está comiendo sus tenedores están bloqueados para el resto. Cuando un filósofo termina de comer este avisa al mesero quién revisa la cola por si alguien estaba esperando por alguno de esos tenedores. La cola se revisa por orden de llegada (FIFO) y de este modo se garantiza que come primero siempre el que primero decidió comer.

Para implementar esto usaremos la clase `Monitor`, qué interactuará sobre los objetos de tipo tenedor, para bloquearlos y desbloquearlos según se estén usando. Entonces cuando un filósofo debe comenzar a comer, se revisa si los tenedores correspondientes no están bloqueado (en uso) y si además la cantidad de gente en la cola esperando por cada uno de estos es cero. Si estas condiciones se cumplen se puede proceder a comer sin problemas, ya que no se afecta a más nadie. Se bloquean entonces los dos tenedores que se usarán. Si este no es el caso entonces esta instancia de filósofo se adiciona a la cola, y además se bloquea la ejecución de su hebra mediante `Monitor.Wait`, mientras espera una señal de `Monitor` correspondiente al objeto `Ticket` (campo de la clase `Philosopher`).

Por otro lado tenemos en la hebra principal (disjunta de las 5 correspondientes a los filósofos) un bucle infinito que en en cada iteración revisa la cola para darle los tenedores libres al que primero los necesite, haciendo `Monitor.Pulse` a su `Ticket` correspondiente. Sin embargo, si se hace esto consecutivamente se caería en un problema de interbloqueo. Además, es innecesario revisar la cola dos veces seguidas si no se han liberado tenedores nuevos. Entonces la ejecución de esta hebra queda bloqueada nuevamente por la clase `Monitor`, aplicada sobre el objeto `Waiter`, hasta que algún filósofo termine de comer y de la señal de desbloqueo.

References

[1] D. D. Anderson and D. F. Anderson: *Factorization in integral domains IV*, Comm. Algebra **38** (2010) 4501–4513.

[2] D. D. Anderson, D. F. Anderson, and M. Zafrullah: *Factorizations in integral domains*, J. Pure Appl. Algebra **69** (1990) 1–19.

[3] D. D. Anderson, D. F. Anderson, and M. Zafrullah: *Factorization in integral domains II*, J. Algebra **152** (1992) 78–93.

[4] D. F. Anderson and D. N. El Abidine: *Factorization in integral domains III*, J. Pure Appl. Algebra **135** (1999) 107–127.

[5] A. Assi and P. A. García-Sánchez: *Numerical Semigroups and Applications*. New York: Springer-Verlag, 2016.
[6] N. R. Baeth and F. Gotti: *Factorization in upper triangular matrices over information semialgebras*, J. Algebra **562** (2020) 466–496.
[7] L. Carlitz: *A characterization of algebraic number fields with class number two*, Proc. Amer. Math Soc., **11** (1960) 391–392.
[8] S. T. Chapman, F. Gotti, and M. Gotti, *Factorization invariants of Puiseux monoids generated by geometric sequences*, Comm. Algebra **48** (2020) 380–396.
[9] S. T. Chapman, F. Gotti, and M. Gotti: *How do elements really factor in* $\mathbb{Z}[\sqrt{-5}]$*?* In: Advances in Commutative Algebra (Eds. A. Badawi and J. Coykendall), pp. 171–195, Springer Trends in Mathematics, Birkhäuser, Singapore, 2019.
[10] S. T. Chapman, F. Gotti, and M. Gotti: *When is a Puiseux monoid atomic?*, Amer. Math. Monthly (to appear). Available on arXiv: https://arxiv.org/pdf/1908.09227.pdf
[11] P. M. Cohn: *Bezout rings and and their subrings*, Proc. Cambridge Philos. Soc. **64** (1968) 251–264.
[12] J. Coykendall and F. Gotti: *On the atomicity of monoid algebras*, J. Algebra **539** (2019) 138–151.
[13] P. A. García-Sánchez and J. C. Rosales: *Numerical Semigroups*, Developments in Mathematics, 20, Springer-Verlag, New York, 2009.
[14] P. A. García-Sánchez and J. C. Rosales: *Numerical semigroups generated by intervals*, Pacific J. Math. **191** (1999), 75–83.
[15] A. Geroldinger and F. Halter-Koch: *Non-Unique Factorizations: Algebraic, Combinatorial and Analytic Theory*, Pure and Applied Mathematics, vol. 278, Chapman & Hall/CRC, Boca Raton, 2006.
[16] F. Gotti: *Atomic and antimatter semigroup algebras with rational exponents*. Available on arXiv: https://arxiv.org/pdf/1801.06779v3.pdf
[17] F. Gotti: *Increasing positive monoids of ordered fields are FF-monoids*, J. Algebra **518** (2019), 40–56.
[18] F. Gotti: *Irreducibility and factorizations in monoid rings*. In: Numerical Semigroups (Eds. V. Barucci, S. T. Chapman, M. D'Anna, and R. Fröberg) pp. 129-139. Springer INdAM Series, Vol. 40, Switzerland, 2020.
[19] F. Gotti: *Puiseux monoids and transfer homomorphisms*, J. Algebra **516** (2018) 95–114.
[20] F. Gotti: *The system of sets of lengths and the elasticity of submonoids of a finite-rank free commutative monoid*, J. Algebra Appl. **19** (2020) 2050137 (2020).
[21] F. Gotti and M. Gotti: *Atomicity and boundedness of monotone Puiseux monoids*, Semigroup Forum **96** (2018) 536–552.
[22] F. Gotti and M. Gotti: *On the molecules of numerical semigroups, Puiseux monoids, and monoid algebras*. In: Numerical Semigroups (Eds. V. Barucci, S. T. Chapman, M. D'Anna, and R. Fröberg) pp. 141–161. Springer INdAM Series, Vol. 40, Switzerland, 2020.
[23] A. Grams: *Atomic rings and the ascending chain condition for principal ideals*. Math. Proc. Cambridge Philos. Soc. **75** (1974), 321–329.
[24] W. Narkiewicz: *Numbers with unique factorization in an algebraic number field*, Acta Arith. **21** (1972) 313–322.
[25] W. Narkiewicz: *On natural numbers having unique factorization in a quadratic number field*, Acta Arith. **12** (1966) 1–22.
[26] W. Narkiewicz: *On natural numbers having unique factorization in a quadratic number field II*, Acta Arith. **13** (1967) 123–129.

Department of Research and Development, Biogen, Cambridge, MA 02142, USA
E-mail address: marly.cormar@biogen.com

Facultad de Matemática y Computación, Universidad de La Habana, San Lázaro y L, Vedado, Habana 4, CP-10400, Cuba
E-mail address: marcosmath44@gmail.com