Cena de filosofos y sincronizacion java

Programación concurrente y Distribuída

Curso 2011-12

Miguel Telleria, Laura Barros, J.M. Drake

`telleriam AT unican.es`

Computadores y Tiempo Real

http://www.ctr.unican.es

Objetivos

- Presentaros la aplicación de “filósofos chinos” con UML
- Practicar lo visto en sincronismo de java
- Prepararos para la practica 2 que es muy similar

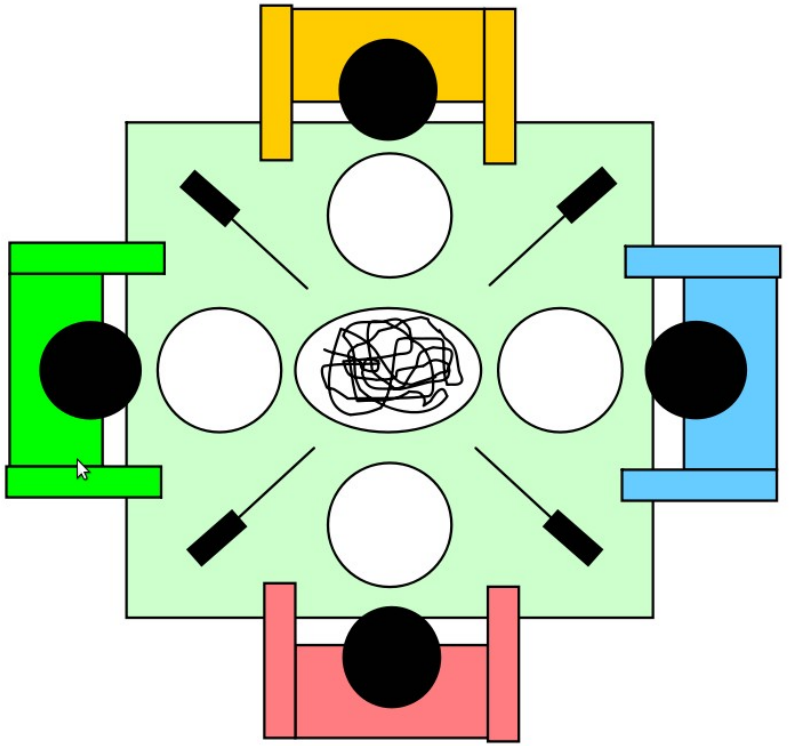
Contenido

- Presentación de los filósofos chinos
- Especificación UML
- Paralelización
- Sincronización de espera
- Sincronización bloqueante

Presentación de los filósofos chinos

Los filósofos chinos

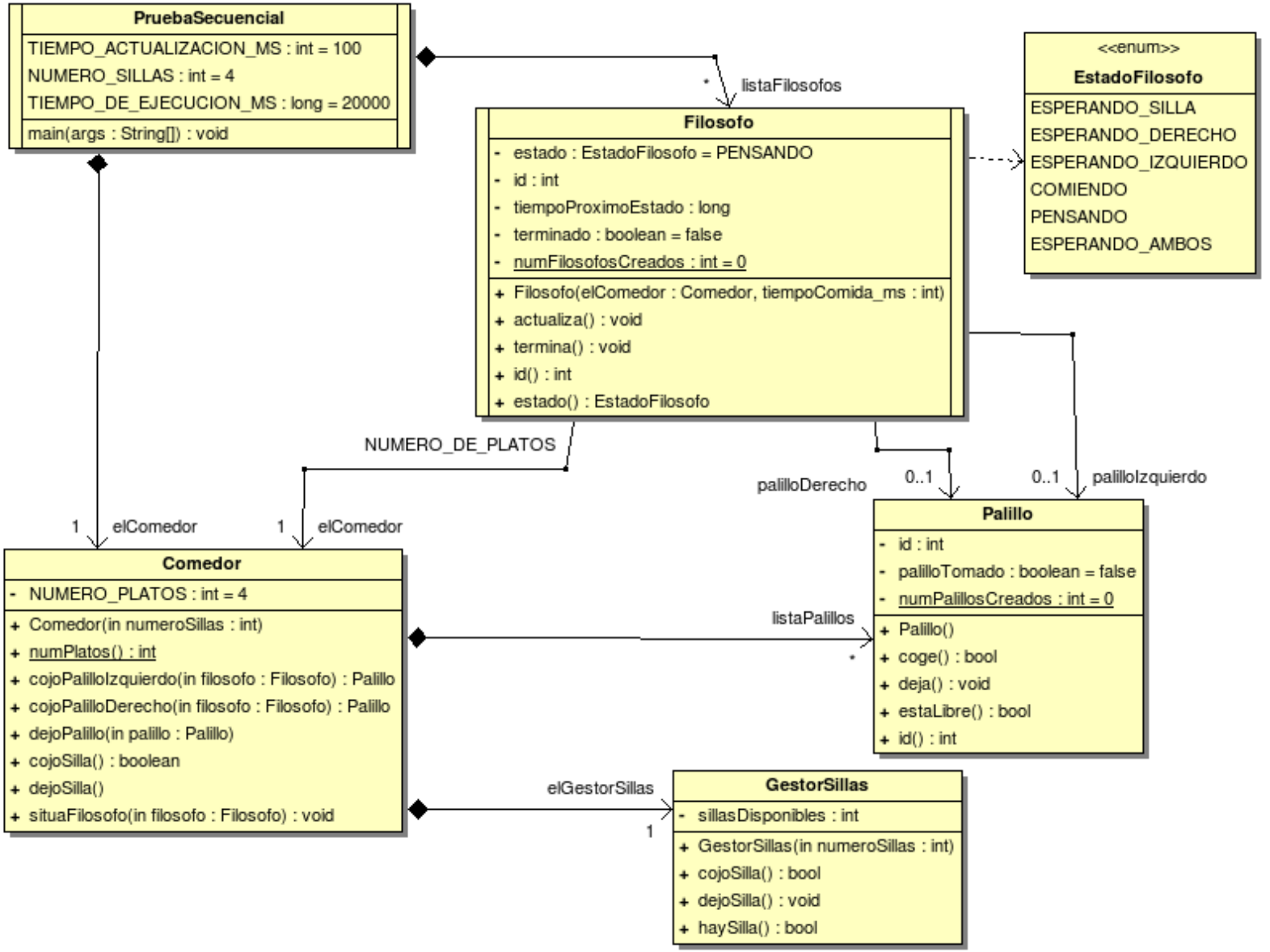
- 4 filósofos
  - Rojo, azul, amarillo, verde
- 4 palillos
  - 1 entre cada 2 filósofos
- Gestor de sillas:
  - Numero de sillas configurable
- Ciclo de cada filosofo:
  - Pensando
  - esperando silla
  - esperando palillo derecho
  - esperando palillo izquierdo
  - comiendo



Vamos a verla funcionar

Especificación UML

Diagrama de clases

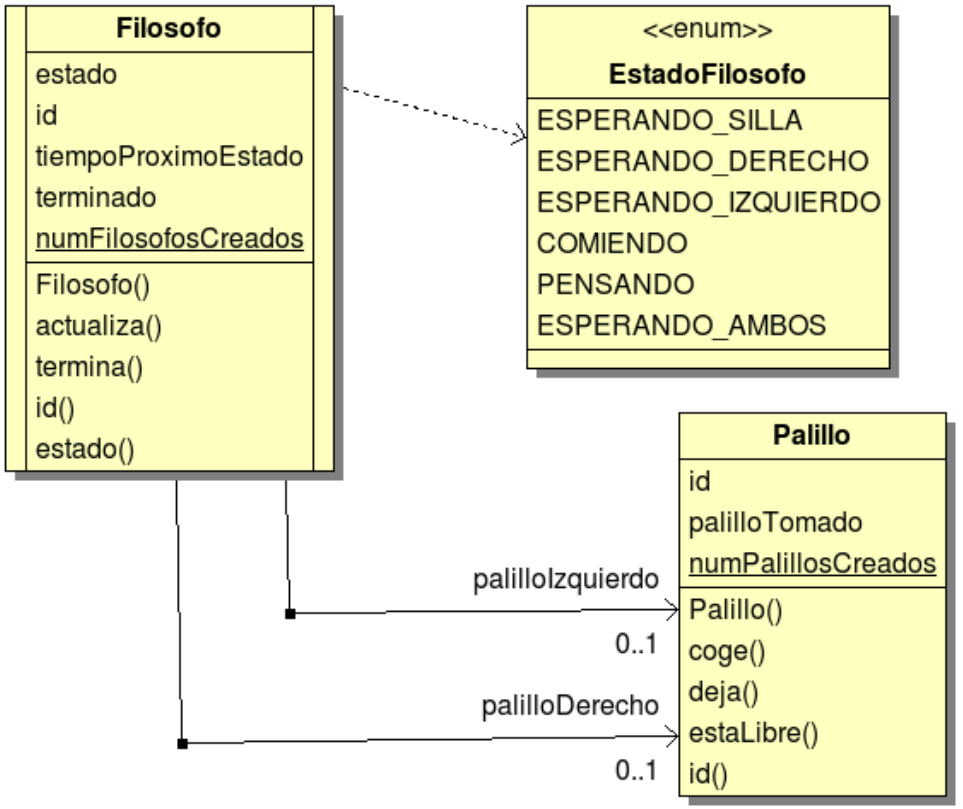


Misiones

- Comedor
  - Datos globales
  - Punto de acceso para el resto de las clases
  - Determina si da una silla o no
  - GUI
- Filosofo
  - Objetos activos
- Gestor de sillas
  - Mantiene la cantidad de sillas libres
- Palillo
  - Modo polling: Devuelve falso si el palillo está cogido
  - Modo espera: Bloquea hasta que el palillo esté libre
- Prueba secuencial
  - Instancia el sistema
  - Crea los threads
  - Termina la ejecución

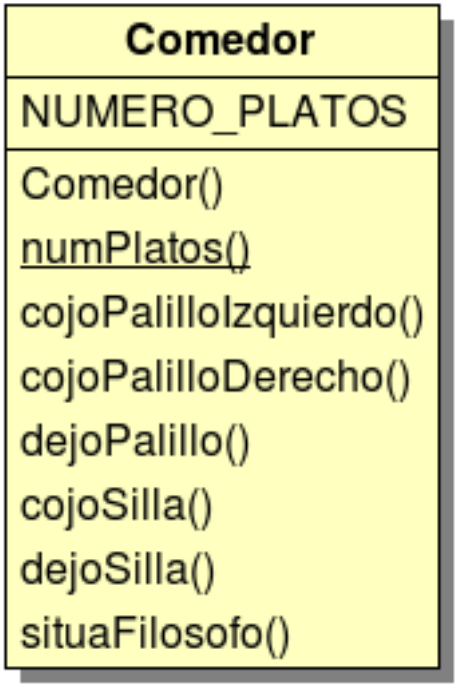
Filosofo

- Objeto activo con su ciclo de vida
- Decisión de diseño:
  - Aunque tiene asociados los palillos, invoca los métodos coge() y deja() a través del comedor



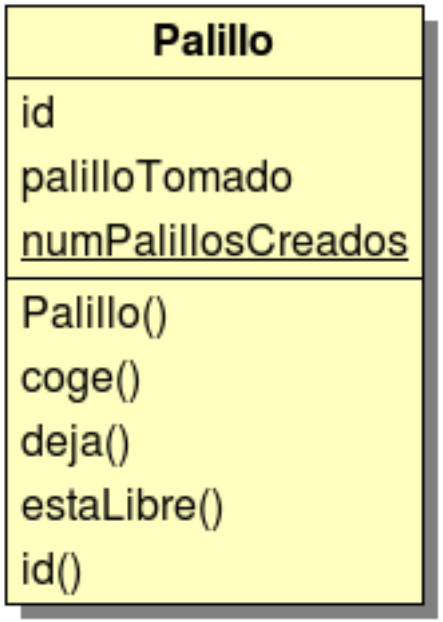
Comedor

- Da acceso al resto de elementos
- Elección de diseño:
  - La clase Palillo devuelve si el palillo está libre
  - El Gestor de Sillas simplemente informa es el comedor el que devuelve si sí o si no.



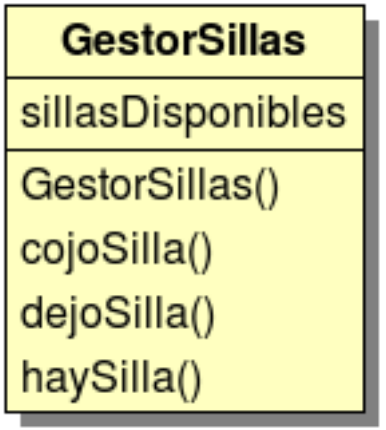
Palillo

- “Se deja” coger o no:
  - Es decir, es el propio palillo el que maneja su política de acceso
  - El atributo privado palilloTomado es el que lleva la cuenta de si hay palillo o no



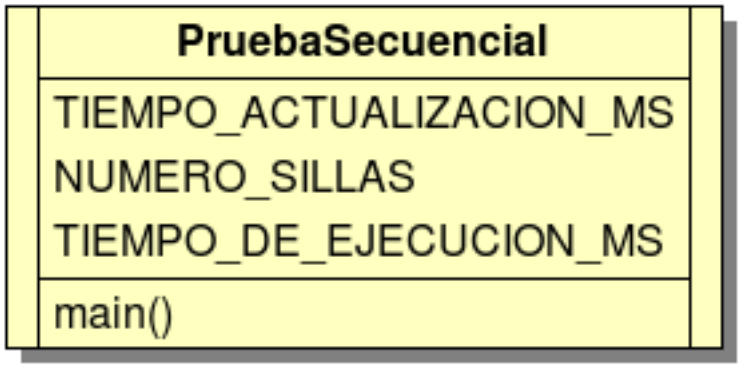
GestorSillas

- Usado para evitar el bloqueo múltiple
  - El cojoSilla() directamenta da el acceso a la silla
  - El atributo privado sillasDisponibles es el que lleva la cuenta de si hay silla libre o no.



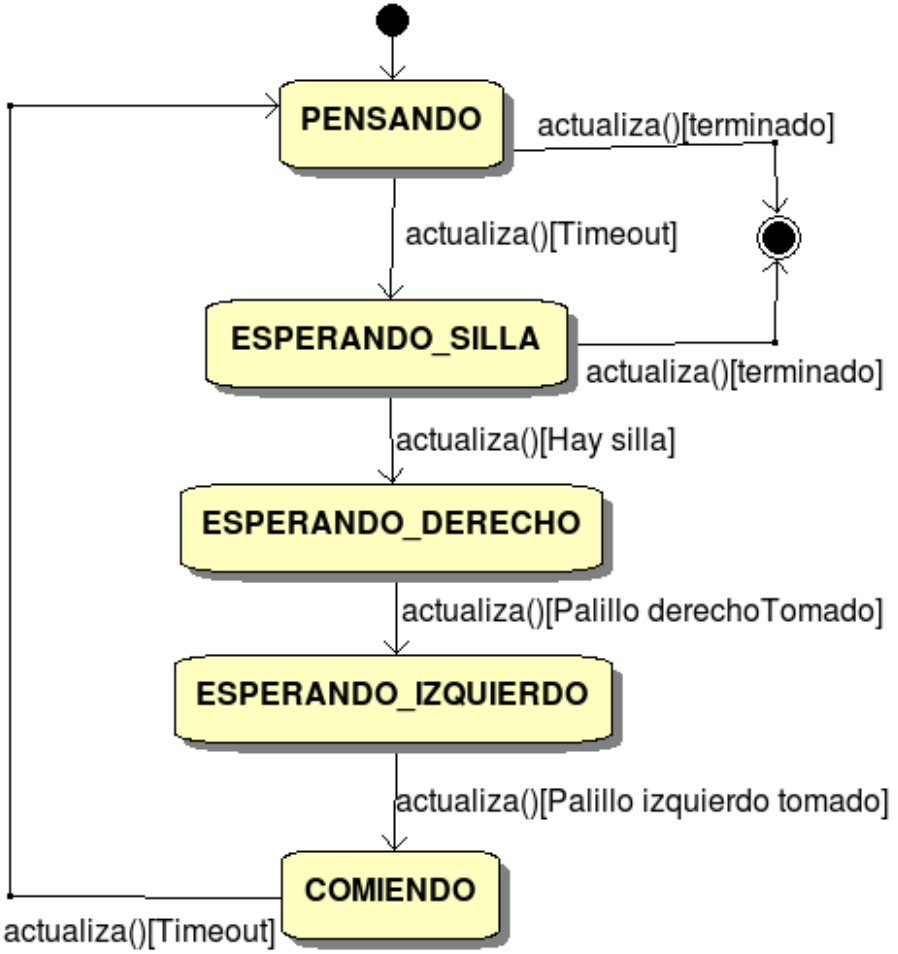
Programa principal

- Instancia los threads y espera a que el tiempo de ejecución se termine.
- Define el numero de sillas
- Define el tiempo de actualización



Estados del filosofo

- El filósofo pasa de un estado a otro mediante el método **actualiza()**.



Paralerización
(similar a la práctica 1)

Cambios en el programa principal

- El main() ya no llama a actualiza
  - Lo hará el propio thread en su run()
- El main se simplifica a:
  - Se crean los filosofos
    - Se les arrancan (si no se arrancan ellos sólos)
  - Se espera un tiempo de ejecución
  - Se ejecuta termina() en ellos

Cambios en el filósofo

- El actualiza() se llama internamente desde el run()
- Se puede auto-arrancar (si heredamos de Thread)
  - Si no nos arranca el que nos crea

Vamos a ejecutarlo

- ¿Se respeta el número de sillas?

- ¿Se mantiene el acceso exclusivo a los palillos?

Sincronización de espera

Identificamos la sincronización

- Que **Objetos** tenemos que sincronizar
  - Información compartida concurrentemente
  - Y que es posible que entre en incoherencia
- Limitación de las **secciones críticas**
  - ¿Lectura?, ¿Escritura?
  - Atomicidad
- **Ámbito** de la sincronización
  - Que objetos tienen que esperar
  - No sobresincronizar

Cosas que no hace falta sincronizar

- Variables locales (ej estado)
  - Ya que están duplicadas en cada thread
- Acceso concurrente en sólo lectura
  - Mientras no haya ningún escritor a la vez claro
- Accesos ya gestionado por una librería o sistema operativo
  - (mientras no importe que se haga en un orden u otro)
  - La GUI: SWT mantiene su propio thread para ello
  - Dispositivos: discos duro, pantalla...

Métodos synchronized vs bloque synchronized

- El método synchronized es equivalente a:
  - Envolver todo el código del método con

    synchronize(this)
- Si la clase está bien diseñada con esto puede bastar
- Nos aseguramos de que “nadie se olvida” sincronizar
  - Con que uno se olvidase no se mantiene la exclusión mutua

Métodos synchronized vs bloque synchronized

- Sin embargo a veces esto no siempre es posible
  - La sincronización (duración o ámbito) puede depender de un argumento, un estado, un instante...
    - Si sé que en cierto estado voy a ser el único en acceder a la información
  - A veces la sección crítica abarca más de un método
    - Ej: test_and_set()
- Buen diseño: poner el mutex lo más profundo posible
  - Los objetos llamantes no deberían verlo
  - La implementación es quien debería definir el ámbito en función de un atributo privado
    - Si es necesario se pone synchronize(this) en parte del código
  - Caso test_and_set(), dar un método sincronizado que englobe los dos.

¿Es necesario sincronizar en la implementación secuencial?

- ¿Si?, ¿No?, ¿Por qué?
  - Y no vale con que “no hay concurrencia” ya que actualiza no deja de ser un planificador cíclico...

¿Es necesario sincronizar en la implementación secuencial? (respuesta)

- NO es necesario
- Puesto que todas las llamadas a actualiza()
  - Se ejecutan de manera atómica
  - Dejan las variables en estado coherente
- Podría no ser así
  - Ej: que haya un estado (miro si silla libre) y luego otro (cojo silla).

Solución primera

- Puesto que todos usamos el comedor
  - Hacemos los métodos del comedor synchronized
- ¿Funciona?
- ¿Que inconvenientes presenta?

Inconvenientes de la solución primera

- Dos filosofos de frente accederían a coge y deja palillo de manera exclusiva.
- El tomar la silla también impediría a otro filósofo empezar a tomar (o dejar el palillo).
- Tenemos “sobre-sincronización” al usar un único lock global

Hilamos más fino

- Cada filosofo retendrá el lock del objeto que esté usando
  - No habrá contención en palillos enfrentados
  - No habrá contención entre palillos y sillas

Sincronización bloqueante

Esperas activa vs espera bloqueante

- Independientemente del ámbito, duración y datos protegidos tenemos 2 estrategias de actuación:
  - Espera activa: Se pregunta reiteradamente el estado de un recurso opcionalmente con un sleep.
    - Ventaja 1: Sencillez de implementación,
    - Ventaja 2: No es bloqueante (ej multicore en ciertos entornos)
    - Inconveniente: Gasto inútil de CPU
  - Espera bloqueante: Se queda dormido a la espera de ser notificado
    - Ventaja: Ahorro de CPU (a veces es relativo)
    - Desventaja: Requiere de ciertos recursos del entorno y de bloqueos
- En la mayoría de los casos preferiremos la espera bloqueante.
  - Ejemplo donde no: Las interrupciones hardware
    - Si nos bloqueamos, el scheduler no nos despierta (no somos proceso del S.O.).
    - Alternativa: Hacemos un pooling y nos volvemos a programar

Como lo hacemos

- Comportamiento de variable de condición + mutex:
  - wait(): Nos suspendemos y **soltamos el lock**
  - notify(): Despertamos a uno y **le entregamos el lock**
  - notifyAll(): Despertamos a todos y se ejecutarán 1 a 1 atómicamente con el lock tomado.
- Cosas a tener en cuenta:
  - Sólo podemos ejecutarlos sobre el objeto en el que estamos sincronizados.
    - Por lo tanto tenemos que estar en ámbito synchronized
  - Siempre que hagáis un wait() no olvidéis que tiene que haber un notify() o notifyAll() correspondiente.
  - Englobar todo en un predicado lógico
    - Ya que al salir del wait() puede seguir habiendo contención y nos tenemos que volver a dormir. Esto ocurre siempre con los notifyAll()

Wait() y sleep()

- Dos diferencias:
  - sleep() no suelta el lock
    - Por lo tanto impedimos el acceso al objeto mientras el objeto duerme.
  - sleep() vuelve inmediatamente de un interrupt()
    - wait() vuelve sólo cuando se le notifica
    - en ambos casos salta la interrupción InterruptedException

¡¡Hagamos la prueba!!

- Diferencias entre palillo y gestor de sillas
  - El objeto palillo puede hacer el wait() el mismo
  - La clase sillas no puede hacer el wait() a menos de que tome la responsabilidad de gestionar su propio acceso.