# Programación Concurrente en Java
# Laboratorio 1: PidTable

J. Blanco, N. Wolovick

## Objetivos

- Solucionar problemas de "thread safety" de una clase, teniendo en cuenta el impacto en la performance.
- Relacionar corrección global de la teoría de Owicki-Gries con algunos problemas de concurrencia típicos (check-then-act, read-modify-write, etc. – *Java Concurrency in Practice*, Chapter 2).

## Actividades

1. Leer y entender el código completo.
   Revisar los `asserts` del código que fueron agregados en la clase `PidTable` siguiendo los lineamientos de Diseño por Contrato.

2. [**20 pts**] Comprobar que las aserciones son localmente válidas, pero no globalmente válidas, mostrando que todo tipo de corrida en entornos secuenciales (`THREAD_NUM` $= 1$) nunca fallan y obteniendo corridas que las invalidan en entornos multihilo (`THREAD_NUM` $> 1$).
   Ayudas: Probablemente haya que usar `java -ea PidTableTest`, donde `ea` significa "*enable assertions*".
   Agregar `StoYield.yield()` en puntos críticos de `PidTable` para generar planificaciones más ricas a fin de invalidar los asserts.
   Comentar todos los `System.out.println()`, pues esto parece producir interleavings mucho más ricos.

3. [**20 pts**] Agregar un `assert` que se debería ser válido al medio del *check-then-act* que aparece en `PidTable.alloc(E)` y ver que **no es globalmente correcto** cuando hay más de un hilo corriendo. Explicar porque dicha aserción es localmente correcta, pero no globalmente correcta.

4. [**30 pts**] Mantener todos los `StoYield.yield()` de la clase `PidTable` que se usaron anteriormente para lograr planificaciones interesantes. Comentar todos los `System.out.println()`, que generan mucho overhead y ocultan las diferencias de performance.
   Sincronizar la clase `PidTable` para evitar que se invaliden los invariantes de representación y la aserción interna de *check-then-act*, asi como de los incrementos de las variables. Usar:

   *a*) Métodos sincronizados.

   *b*) Bloques sincronizados solo en el lugar donde resulten necesarios.

   *c*) `AtomicInteger` para los contadores `size` y `capacity` y bloques sincronizados para las otras "condiciones de carrera".

   Generar tres clases nuevas (`PidTable1`, `PidTable2`, `PidTable3`) cada una en un archivo separado.

5. [**30 pts**] Medir e informar la performance de cada uno de los métodos de sincronización utilizados anteriormente.
   Ayudas: Usar el comando **NIX* `time java -ea PidTableTest >/dev/null`. Calibrar `PidTableUse.CYCLE_NUM` para que la ejecución tome un tiempo significativo (2 o 3 segundos) a fin de enmascarar lo más posible el efecto de caches frías, tiempos de carga, optimizaciones runtime, etc. . Repetir el experimento cierta cantidad de veces a fin de promediar y mitigar el ruido del sistema en general.

## Actividades Extra

1. [**10 pts**] Explicar porque `System.out.println()` impiden la aparición de interleavings ricos.
2. [**10 pts**] Existen dos scopes donde es posible sincronizar el lazo el método `Pidtable.alloc(E)`, uno interno y otro externo. Medir la diferencia de performance de cada uno y explicar los resultados.
3. [**10 pts**] Utilizar `ConcurrentMap` para implementar la clase. Comparar la performance respecto a los puntos anteriores.