Práctica 2: Sincronización de hilos

2018 – Sistemas Operativos II

Licenciatura en Ciencias de la Computación

Entrega: lunes 16 de abril

Nota: debe utilizar el Subversion de la materia creando un subdirectorio por alumno/grupo en:
`https://svn.dcc.fceia.unr.edu.ar/svn-no-anon/lcc/R-412/Alumnos/2018/`

1. Introducción

Al resolver los ejercicios, recuerde que el código bien sincronizado debe funcionar sin importar qué orden elija el planificador para ejecutar los hilos listos. Más explícitamente: se debería poder poner una llamada a `Thread::Yield` en cualquier parte del código donde las interrupciones estén habilitadas sin afectar la corrección del código. Se recomienda usar la opción `-rs` de Nachos.

2. Ejercicios

1. Implemente "locks" (candados) y variables de condición. Use semáforos como base; esto implica que tanto los "locks" como las variables de condición **no** deben apagar las interrupciones ni dormir hilos, sino proveer su funcionalidad por medio de semáforos.

   En `threads/synch.hh` están las interfaces públicas, en las clases `Lock` y `Condition`. Se deben definir los datos privados e implementar la interfaz. Debe implementar también la función `Lock::IsHeldByCurrentThread` y utilizarla para comprobar (mediante `ASSERT`) que el hilo que realice un `Acquire` no posea el "lock" y que el hilo que haga `Release` sí lo posea.

2. Implemente paso de mensajes entre hilos a través de puertos, que permitan que los emisores se sincronicen con los receptores. Haga una nueva clase `Port` con los siguientes métodos:

   ```
   void Port::Send(int message);
   void Port::Receive(int *message);
   ```

   `Send` espera atómicamente hasta que se llame a `Receive` y luego copia el mensaje en el búfer de `Receive`. Una vez hecha la copia, ambos pueden

retornar. La llamada a `Receive` también es bloqueante: en caso de que no haya ningún emisor esperando, espera a que llegue uno (es decir, que se ejecute un `Send`).

La solución debe funcionar incluso si hay múltiples emisores y receptores para el mismo puerto.

3. Implemente un método `Thread::Join` que bloquee al llamante hasta que el hilo en cuestión termine.

```
Thread *t = new Thread("Hijo");
t->Fork(func, 0);
t->Join();  // Acá el hilo en ejecución se bloquea
            // hasta que 't' termine.
```

Agregue un argumento al constructor de `Thread` que indique si se llamará a `Join` sobre este hilo.

La solución debe borrar adecuadamente el bloque de control del hilo ("thread control block" o TCB), tanto si se hará `Join` como si no y aunque el hilo hijo termine antes de la llamada a `Join`.

4. *a*) El planificador de Nachos implementa una política de "round robin". Implemente multicolas con prioridad.

   Establezca prioridades fijas para cada hilo (positivas, 0 es la menor prioridad). El planificador debe elegir siempre el hilo listo con mayor prioridad.

   b) Modifique la implementación para solucionar o evitar en el caso de los "locks" y variables de condición el problema de inversión de prioridades.

   Explique (en un archivo de texto) por qué no puede hacerse lo mismo con los semáforos.