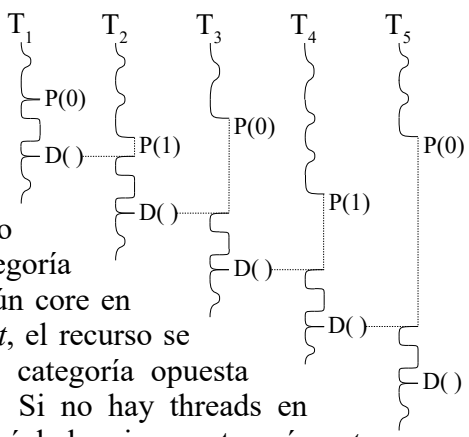
El siguiente es el mismo problema del control. Esta vez debe resolverlo usando spin-locks.

Se dispone de un recurso único compartido por los múltiples cores de una máquina *sin sistema operativo*. Los cores se agrupan en categoría 0 y categoría 1. Se dice que 0 es la categoría opuesta de 1, y 1 la categoría opuesta de 0. Un core de categoría *cat* solicita el recurso invocando la función *pedir(cat) y d*evuelve el recurso llamando a la función *devolver()*, sin parámetros.



Se necesita programar las funciones *pedir* y *devolver* garantizando la exclusión mutua al acceder al recurso compartido. Se requiere una política de asignación alternada primero y luego por orden de llegada. Esto significa que cuando un core de categoría *cat* devuelve el recurso, si hay algún core en espera de la categoría opuesta a *cat*, el recurso se asigna inmediatamente al core de categoría opuesta que lleva más tiempo esperando. Si no hay threads en espera de la categoría opuesta pero sí de la misma categoría *cat*, el recurso se asigna al core que lleva más tiempo en espera. Si no hay ningún core en espera, el recurso queda disponible y se asignará en el futuro al primer core que lo solicite, cualquiera sea su categoría. El diagrama de arriba muestra un ejemplo de asignación del recurso. La invocación de *pedir* se abrevió como P(…) y la de *devolver* como D( ). Los nombres T1, T2, ..., etc. son solo referenciales. No significa que corran en el core 1, core 2, ..., etc.

Programe las funciones *pedir, devolver*, *iniciar* y *terminar* con encabezados:

```
void iniciar();
void terminar();
void pedir(int cat);
void devolver();
```

Dado que la máquina no posee un sistema operativo, la única herramienta de sincronización disponible son los spin-locks. Ud. sí dispone de las *fifoqueues*. También dispone de *coreId()* para determinar la identificación del core, pero no le será útil en esta tarea. No necesita saber que la máquina tiene 33 cores.

Dado que no hay sistema operativo, no puede usar funciones como *nMalloc*, *nMakeMonitor*, *nMakeSem*, etc. Puede usar *nPrintf* para fines de depuración pero no olvide eliminarlos antes de entregar su tarea.

Inicialice las variables globales en la función *iniciar* y libere recursos solicitados en *terminar* (como fifoqueues y spin-locks).

Instrucciones

Descargue de U-Cursos *t4.zip* y descomprímalo. El directorio T4 contiene: una plantilla para su solución en el archivo *t4.c.plantilla*; los encabezados de las funciones pedidas en *pedir.h*; el programa de prueba en *test.c*, y; el archivo *Makefile* para que compile su tarea. Programe su solución en el archivo *t4.c*, basándose en *t4.c.plantilla*.

Para depurar con ddd compile con: *make clean; make test-t4; ddd test-t4*

Depure errores de manejo de memoria con: *make run-valgrind-ddd*

Depure potenciales dataraces con: *make run-drd-ddd*

Compile y ejecute con opciones de optimización con: *make run-O*

Considerando que sería muy difícil probar de verdad su tarea en una máquina sin sistema operativo, pruebe su tarea usando *psystem64-beta3* (esta última versión solo resuelve una gotera de memoria en *psystem64-beta2*). El programa de prueba hace uso de *psystem64* para implementar spin-locks a partir de los monitores de pSystem/nSystem y por lo tanto no son verdaderos spin-locks, pero sí son casi funcionalmente equivalentes. La API para trabajar con estos spin-locks está en *spinlock.h*. La siguiente tabla muestra las diferencias de uso entre los verdaderos spin-locks y los que Ud. deberá usar:

verdaderos spin-locks	*spin-locks en su tarea*
`int sl= OPEN; // o CLOSED`	`SpinLock sl;`
	`initSpinLock(&sl, OPEN);`
`spinLock(&sl);`	`spinLock(&sl);`
`spinUnlock(&sl);`	`spinUnlock(&sl);`
	`destroySpinLock(&sl);`

Observe que Ud. debe inicializar estos spin-locks y destruirlos para evitar goteras de memoria.

Entrega

Ud. debe entregar por medio de U-cursos el archivo *t4.c*. No se aceptarán tareas que no funcionen con el test de prueba. Se descontará medio punto por día hábil de atraso.