CC4302 Sistemas Operativos – Control 2 – Semestre Otoño 2017 – Prof.: Luis Mateu

Pregunta 1

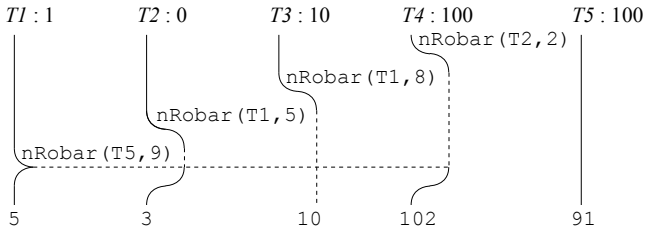
A) (4 puntos) Considere que una tarea en nSystem almacena inicialmente 100 euros. Una tarea puede robar dinero desde otra tarea con la siguiente función:

void nRobar(nTask desde, int cantidad);

En tal caso se le resta *cantidad* euros a la tarea *desde* y se le suman a la tarea que invocó *nRobar*. El parámetro *cantidad* es siempre mayor que cero. Un primer invariante es que una tarea no puede almacenar una cantidad negativa de euros. Si la tarea *desde* no posee suficiente dinero para robarle *cantidad* entonces *nRobar* debe esperar hasta que la tarea *desde* sí posea al menos la cantidad requerida. El segundo invariante es que en un instante dado la tarea *T* no puede estar esperando robarle *c* euros a la tarea *U* si *U* tiene al menos *c* euros.

Observe que cuando la tarea *T* lograr robar dinero, varias otras tareas pueden estar esperando poder robarle dinero a *T*. No está especificado en qué orden deben robarle el dinero a *T*.

El siguiente diagrama de tareas es un ejemplo de uso de *nRobar*.



En el inicio del diagrama las 5 tareas T1 a T5 poseen 1, 0, 10, 100 y 100 euros respectivamente. Al final poseen 5, 3, 10, 102 y 91 euros. La tarea T4 trata de robarle a T2, luego T2 y T3 tratan de robarle a T1. Como sus víctimas no tienen dinero suficiente deben esperar. Cuando T1 roba 9 euros, T2 logra su robo y por lo tanto T4 también lo logra, pero T3 continúa esperando. Note que es válido que T3 hubiese logrado su robo y dejar esperando a T2 (y T4).

Programe la función *nRobar* usando las funciones de bajo nivel de nSystem (*START_CRITICAL*, *Resume*, *PutTask*, etc.). Indique qué nuevos campos necesita agregar al descriptor de tarea y su valor inicial. Ud. *no puede* usar otros mecanismos de sincronización ya disponibles en nSystem, como semáforos, monitores, mensajes, etc.

B) (2 puntos) Haga una tabla con 3 columnas para las estrategias de scheduling *first come first served*, *shortest job first* y *round robin*, y 4 filas para las propiedades (i) tiempo de despacho, (ii) tiempo de respuesta, (iii) simplicidad de la implementación, y (iv) número de cambios de contexto implícitos. En las celdas coloque palabras como malo, bueno, regular, pésimo, simple, compleja, 0, pocos, muchos, etc. para indicar cómo le va a una estrategia comparada con las demás.

Pregunta 2

I.- (4 puntos) Considere una máquina con 8 cores físicos que comparten la memoria, sin un núcleo de sistema operativo y por lo tanto no hay scheduler de procesos. De manera similar a la pregunta 1 a cada core se le asignan inicialmente 100 euros. Un core puede robar *cantidad* euros del core número *desde* invocando la función:

void robar(int desde, int cantidad);

Programe la función *robar* y declare las variables globales que necesite señalando su valor inicial. Ud. dispone de la función *coreId( )* que entrega el número del core que la invoca (entre 0 y 7). Use *spin-locks* para sincronizar los distintos cores.

Restricciones: No hay: *nMalloc, fifoqueues*, *LLMutex*, cola de procesos *"ready"*. Para esperar no le queda otra que hacerlo mediante un *spin-lock*, pero no puede recurrir a otra forma de *busy-waiting*.

Ayuda: Use una matriz *m* de 8 por 8 punteros a spin-locks. Si *m[i][j]* no es nulo quiere decir que *m[i][j]* es la dirección de un *spin-lock* en el que el core *i* espera para robarle al core *j*.

II.- (1 punto) Considere 3 tipos de núcleos: un núcleo clásico de Unix para una máquina mono-core, un núcleo moderno para una máquina mono-core y un núcleo moderno para una máquina multi-core. ¿Cómo se garantiza la exclusión mutua al acceder a la cola de procesos "ready" para cada uno de ellos y por qué?

III.- (1 punto) ¿Cómo se evita en Unix que un proceso del usuario tenga acceso a la cola de procesos "ready", a los descriptores de proceso, al vector de interrupciones, etc.? ¿Pero si no tiene acceso, cómo logra la función *fork* crear un nuevo descriptor de proceso y agregarlo a la cola de procesos "ready"? Explique.