UNIVERSIDAD DEL BÍO-BÍO

GUIA DE EJERCICIOS

Sistemas Operativos – Otoño 2018

Profesor Luis Gajardo

INGENIERÍA CIVIL INFORMÁTICA
UNIVERSIDAD DEL BÍO-BÍO

1. Conceptos de nSystem

   a) ¿Para qué sirven los procedimientos START_CRITICAL y END_CRITICAL en nSystem?

   b) ¿Quién los usa: el programador que usa nSystem o el programador que implementa nSystem?

   c) nSystem es un sistema de procesos livianos que se implementan dentro de un proceso Unix. ¿Por qué no es posible implementar procesos pesados dentro de un proceso Unix?

2. Se desea agregar a la implementación que actual de nSystem un nuevo mecanismo de sincronización entre procesos. Este mecanismo esta basado en *condiciones*, que se manipulan con los siguientes procedimientos:

   - `Condition MakeCondition()`: Construye y retorna una nueva condición.

   - `void WaitCondition(Condition cond)`: Bloquea el proceso que lo invoca hasta que otro proceso invoque en el futuro *NofityCondition* sobre *cond*. Observe que este procedimiento siempre se bloquea, sin importar los *NotifyCondition* que se hayan hecho en el pasado.

   - `void NotifyCondition(Condition cond)`: Desbloquea todos los procesos que esperaban la condición *cond* y retorna de inmediato. El orden en que se retoman los procesos puede ser cualquiera.

   Implemente estos procedimientos en nSystem. Suponga que nSystem no posee semáforos, ni mensajes ni monitores y por lo tanto deberá basar su solución en la API de nivel implementador para responder esta pregunta.

3. Implemente los procedimientos *nExitTask* y *nWaitTask* de nSystem. La descripción de estos procesos es la siguiente:

   - `void nExitTask(int rc)`: Termina la ejecución de la tarea que lo invoca, *rc* es el código de retorno de la tarea.

   - `int nWaitTask(nTask task)`: Espera a que una tarea termine, entrega el código de retorno dado a *nExitTask*.

   Recuerde que cada *nExitTask* va asociando a un y solo un *nWaitTask*. El orden en que se invocan estos dos procedimientos es indeterminado. Suponga que el programador los utiliza correctamente y por lo tanto no necesita verificar que se invoquen una sola vez.

   Para implementar estos procedimientos Ud. debe usar los siguientes procedimientos de bajo nivel de nSystem:

- `PushTask(ready_queue, task)`: Permite agregar una tarea a la cola de tareas listas para ejecutarse.

- `ResumeNextReadyTask()`: Extrae la primera tarea en la cola de tareas listas para ejecutarse y le transfiere el procesador.

- `DestroyTask(task)`: Suponga que este procedimiento libera los recursos que ocupa una tarea, como la pila y el descriptor de tarea (este procedimiento no existe realmente en nSystem, pero sirve para simplificar esta pregunta).

Además Ud. puede usar como estime conveniente los campos *wait_task* (de tipo *nTask*) y *rc* (de tipo *int*) presentes en el descriptor de tarea.

4. Un *lock* es una abstracción que permite sincronizar tareas. Un *lock* puede ser poseído a lo más por una tarea en un instante dado. Los procedimientos para manipular *locks* son:

   - `LOCK MakeLock()`: Crea y entrega un lock.

   - `void DestroyLock(LOCK lock)`: Destruye lock.

   - `void AcquireLock(LOCK lock)`: Solicita lock en exclusividad. La tarea se bloquea temporalmente si lock estaba en posesión de otra tarea.

   - `void ReleaseLock(LOCK lock)`: Libera lock. Si habían otras tareas en espera de lock, puede continuar aquella que pidió primero el lock.

5. En este ejercicio se debe implementar pipes para el nSystem. Los pipes son una forma de comunicación asíncrona e indirecta entre procesos (como los pipes de Unix). Los pipes se manejarán con los siguientes procedimientos:

   - `nPipe MakePipe(int buffer_size)`: Crea un nPipe con un buffer interno de tamaño *buffer_size*.

   - `char nGetPipe(nPipe pipe)`: Recupera un carácter desde el buffer. En caso que no hayan caracteres disponibles se debe bloquear hasta recibir uno.

   - `void nPutPipe(nPipe pipe, char c)`: Agrega al buffer un carácter. En caso que hayan tareas esperando, debe despertar alguna. En caso que el buffer esté lleno debe esperar hasta que haya un espacio disponible.

   - `void nClosePipe(nPipe pipe)`: Destruye y libera el nPipe. Sólo puede invocarse si no hay tareas emisoras o receptoras en espera.

   Como simplificación, no es necesario que el pipe maneje un estado abierto/cerrado. El programa cliente esperará por productores y consumidores antes de invocar a nClosePipe. Utilice los procedimientos de bajo nivel disponibles en nSystem.