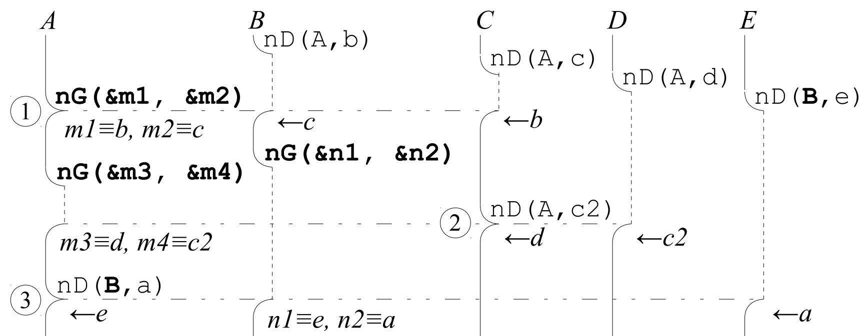
CC4302 Sistemas Operativos – Control 2 – Semestre Otoño 2014 – Prof.: Luis Mateu

Pregunta 1

Se necesita agregar la siguiente API a nSystem para la recepción de 2 mensajes simultáneos:

- `void *nDeliver(nTask t, void *msg)`: Envía el mensaje *msg* a la tarea *t*. Esta función solo retorna cuando la tarea *t* recibe el mensaje con *nGather*, entregando como resultado el mensaje de la otra tarea que también invocó *nDeliver*.
- `void nGather(void **pmsg1, void **pmsg2)`: Recibe los mensajes enviados por otras 2 tareas con *nDeliver(t, m)*, considerando que *t* es la tarea que invocó *nGather*. Los mensajes de ambas tareas quedan almacenados en **pmsg1* y **pmsg2*. Si en el momento de invocar *nGather* no hay 2 invocaciones pendientes de *nDeliver*, entonces *nGather* espera.

El siguiente diagrama muestra un ejemplo de ejecución:



Se abrevió *nGather* por nG y *nDeliver* por nD. Observe que en 1 se concreta de inmediato el *nGather* de *A* con mensajes (*nDeliver*) emitidos previamente por *B* y *C*. Note también que la llamada a *nDeliver* de *B* retorna el mensaje *c* y el de *C* retorna *b*. En 2 el mensaje de C concreta el *nGather* iniciado previamente por *A* y el mensaje de *D*. En 3 el mensaje de A concreta el *nGather* iniciado previamente por *B* y el mensaje de *E*.

Implemente esta API usando los procedimientos de bajo nivel de nSystem (*START_CRITICAL*, *Resume*, *PutTask*, etc.). Ud. *no puede usar* otros mecanismos de sincronización ya disponibles en nSystem, como semáforos, monitores, mensajes, etc. Ud. necesitará agregar nuevos campos al descriptor de tarea de nSystem como un cola para las tareas que esperan en un *nDeliver*, un contador de tareas que esperan en nDeliver, el mensaje suministrado mediante *nDeliver*, el mensaje que debe retornar *nDeliver*, etc. No se preocupe por el orden en que se retoman los procesos después de concretar un *nGather* (no necesita ser estrictamente *last come first served*).

Pregunta 2

Parte a.- (1 punto) Una ráfaga de un proceso se ejecuta en 3 tajadas (*slices*). ¿De qué tipo de scheduling se trata? ¿Preemptive o non-preemptive? Explique.

Parte b.- (1 punto) ¿Cuantos spin-locks se necesitan en un núcleo clásico de Unix para una máquina multi-core? ¿Y si es mono-core?

Parte c.- (4 puntos) Se tiene una máquina *octa-core* en la que no existe un núcleo de sistema operativo y por lo tanto solo se pueden usar los *spin-locks* para sincronizar los cores. Se le pide implementar un semáforo en donde la prioridad de entrega de los tickets está dada por el número del core que solicita el ticket. El core 0 tiene la mejor prioridad y el core 7 la peor. La función *coreId()* entrega el número del core.

Concretamente Ud. debe definir el tipo *Sem* y programar las siguientes funciones:

`void iniSem(Sem *sem, int ini);`	Inicializa un semáforo con *ini* tickets disponibles
`void waitSem(Sem *sem);`	Solicita un ticket
`void signalSem(Sem *sem);`	Deposita un ticket

Dado que no hay núcleo de sistema operativo Ud. no dispone de monitores o semáforos y tampoco puede usar funciones como *Resume*, *PutTask*, etc. Ni siquiera existe *malloc* ni el tipo cola de prioridades. Por la misma razón, cuando un core invoca *waitSem* y no hay tickets disponibles, no queda otra alternativa que esperar en un ciclo de *busy-waiting*, por ejemplo usando un spin-lock.

Hint: Use un arreglo de 8 elementos, uno por cada core. Cada elemento almacena la información que Ud. necesite para determinar si ese core está esperando o no un ticket. Cuando se deposite un ticket recorra secuencialmente el arreglo para otorgarlo al core en espera que tenga la mejor prioridad.