Pregunta 1

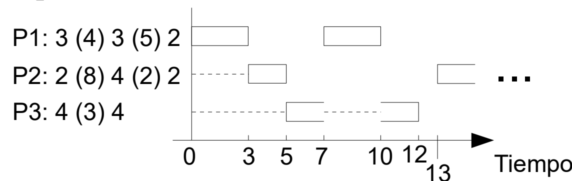
I) (4 puntos) Se desea agregar un sistema de mutex con prioridades a nSystem como herramienta de sincronización nativa. Concretamente la API de este sistema es:

- *nMutex *nMakePriMutex( )*: Entrega un mutex con prioridades. En todo instante puede existir a lo más un solo propietario de este mutex.
- *void nLock(nMutex *mtx, int pri)*: Solicita la propiedad de *mtx* con prioridad *pri* (un valor entero entre 0 y 3). Si *mtx* está libre, el solicitante adquiere su propiedad de inmediato y *nLock* retorna. Si actualmente otra tarea tiene la propiedad de *mtx*, el solicitante espera hasta adquirir en forma exclusiva *mtx* cuando otra tarea invoque *nUnlock*.
- *void nUnlock(nMutex *mtx)*: Libera *mtx*. Si existen varias tareas esperando adquirir *mtx*, se entrega su propiedad a la tarea que lo haya solicitado con la prioridad más alta (3 es la más alta, 0 la menor).

Defina la estructura de datos para *nMutex* 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.

Ayuda: Use 4 colas, una para cada prioridad posible. Cuando se solicite con prioridad *k* un mutex que está ocupado, haga que ese thread espere en la cola *k*. Al liberar el mutex recorra las colas por orden de prioridad. Si encuentra una cola no vacía, otorgue el mutex al thread que está en primer lugar en esa cola.

II) (2 puntos) El diagrama que viene a continuación muestra el scheduling de 3 procesos.



En tiempo 0 los 3 procesos están READY (línea punteada). La estrategia de scheduling es en base a prioridades fijas y distintas. Junto a cada proceso se indica la duración de las ráfagas de CPU y entre paréntesis la duración de los estados de espera. Responda: **a.-** Ordene los procesos de mejor a peor prioridad (0,5 puntos) **b.-** Explique si se trata de scheduling *preemptive* o *non-preemptive* (0,5 puntos) **c.-** Complete el diagrama (1 punto).

Pregunta 2

A) (4 puntos) Resuelva el mismo problema de la pregunta 1 considerando ahora una máquina *octa-core* en la que no existe un núcleo de sistema operativo y por lo tanto no hay un scheduler de procesos. La estructura de datos y las funciones que Ud. debe programar son las siguientes:

```
typedef struct { … } Mutex;
void initMutex(Mutex *mtx);
void lock(Mutex *mtx);
void unlock(Mutex *mtx);
```

La función *lock* no recibe un parámetro para la prioridad porque esta está dada por el número del core que solicita la operación. El core 0 tiene la mejor prioridad y el 7 la peor. Para programar su solución Ud. dispone de *spin-locks* y la función *coreId( )* que entrega el número del core que la invoca.

Restricción: Dado que no hay un núcleo de sistema operativo, la única forma válida de esperar a que el mutex se libere es utilizando un spin-lock. Otras formas de *busy-waiting* no están permitidas. No hay *malloc* ni *fifoqueues*.

Ayuda: Use un arreglo de 8 punteros a spin-locks. Cuando el core *k* solicite un mutex que está ocupado, haga esperar ese core en un spin-lock y coloque la dirección del spin-lock en el *k*-ésimo elemento del arreglo. Cuando se libere el mutex, recorra ascendentemente el arreglo. Si encuentra un elemento no nulo, es la dirección de un spin-lock en donde espera el core al que le debe otorgar el mutex.

B) (1 punto) ¿Cuantos spin-locks se necesitan en un núcleo clásico de Unix para una máquina multi-core? ¿Y cuantos spin-locks se necesitan en un núcleo moderno para una máquina mono-core? Explique.

C) (1 punto) Ud. debe elegir una herramienta para garantizar la exclusión mutua en una sección crítica de un núcleo moderno de Unix para una máquina multicore. ¿Bajo qué condiciones sería más eficiente usar un spin-lock y cuando sería más eficiente un semáforo?