# CC4302 Sistemas Operativos – Examen I – Semestre Otoño 2020 – Prof.: Luis Mateu

## Pregunta 1

Se desea agregar *buffers* en forma nativa a nSystem. Estos se usarán para sincronizar múltiples productores y consumidores. Concretamente Ud. debe definir la estructura *nBuffer* y programar las siguientes funciones:

- *nBuffer *nMakeBuffer( )*: Crea y retorna un buffer de tamaño ilimitado.
- *void nPut(nBuffer *b, void *ptr)*: Deposita el ítem *ptr* en el buffer *b*. Esta función nunca espera porque el buffer es de tamaño ilimitado. Si hay tareas en espera en un *nGet* del mismo buffer *b*, el ítem se otorga de inmediato a aquel que lleva más tiempo esperando.
- *void *nGet(nBuffer *b)*: Extrae y retorna un ítem del buffer *b*. Si el buffer está vacío espera hasta que se le otorgue un ítem con *nPut*.

***Restricciones***: Ud. debe usar los procedimientos de bajo nivel de nSystem (*START_CRITICAL*, *Resume*, *PutTask*, etc.). No puede usar otros mecanismos de sincronización ya disponibles en nSystem, como semáforos, monitores, mensajes, etc. Para almacenar los ítemes debe usar una *fifoqueue* (no use un arreglo circular).

## Pregunta 2

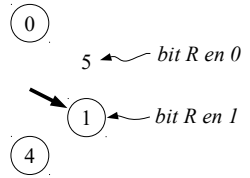
Resuelva el mismo problema de la pregunta 1 considerando ahora una máquina *octa-core* en la que no existe un núcleo de sistema operativo y por lo tanto no hay un scheduler de procesos. La estructura de datos que debe definir y las funciones que debe programar son las siguientes:

```
typedef struct { … } Buffer;
void *initBuffer(Buffer *b);
void put(Buffer *b, void *ptr);
void *get(Buffer *b);
```

***Restricciones***: Dado que no hay un núcleo de sistema operativo la única forma válida de garantizar la exclusión mutua es usando un spin-lock. Para esperar hasta que se otorgue un ítem a *get* debe recurrir a un spin-lock inicialmente cerrado. Otras formas de *busy-waiting* no están permitidas. Use una *fifoqueue* para almacenar los ítemes y otra *fifoqueue* para almacenar las direcciones de los spin-locks en donde esperan las llamadas a *get* pendientes. Los ítemes deben ser otorgados a los *get* por orden de llegada.

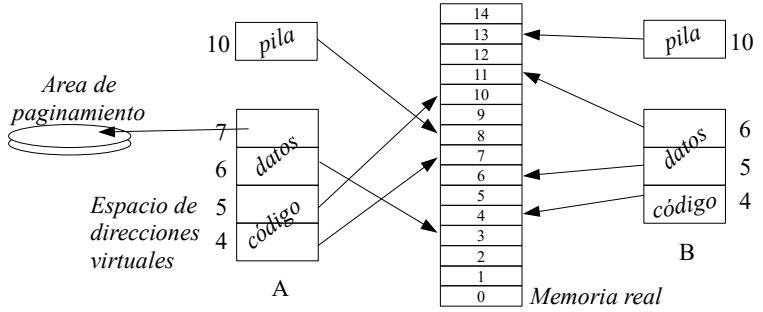
## Pregunta 3

**A)** Considere un sistema Unix que implementa la estrategia del reloj. El sistema posee 6 páginas reales disponibles y corre un solo proceso. La figura indica el estado inicial de la memoria, mostrando las páginas residentes en memoria, la posición del cursor y el valor del bit R.




Dibuje los estados por los que pasa la memoria para la siguiente traza de accesos a páginas virtuales: 8 4 6 1 0 2

**B)** El diagrama de más abajo muestra la asignación de páginas en un sistema Unix que ejecuta los procesos A y B. Las páginas son de 4 KB. El núcleo utiliza la estrategia *copy-on-write* para implementar *fork*. (a) Construya la tabla de páginas del proceso A después de que este invoca *sbrk* pidiendo 6 KB adicionales. (b) Considere que el proceso B invocó *fork*. Construya la tabla de páginas para el proceso hijo justo después de que este modificó la página 5. No construya la tabla del padre. En las tablas indique página virtual, página real y atributos de validez y escritura.




**C)** Las variables globales son siempre una fuente de dataraces en los programas multi-thread. Explique cómo se resuelve el problema de los dataraces que podría producir las variables globales declaradas en un núcleo clásico de Unix. ¿Y en un núcleo moderno?

**D)** Un programador propone hackear un sistema Unix de la siguiente manera. El sabe en qué dirección de la memoria está el vector de interrupciones. Entonces él propone cambiar la función que ejecuta las llamadas al sistema por su propia función. Explique si puede o no lograr hackear la máquina.