Pregunta 1

El siguiente código es una implementación incorrecta e ineficiente de un sistema de subastas para una máquina octa-core, sin un núcleo de sistema operativo y por lo tanto no hay scheduler de procesos.

```
typedef struct {
  PriQueue c; // cola de prioridades
  int n;      // número de unidades
} *Subasta;
typedef struct {
  int oferta; // dinero ofrecido
  int resol; // estado de la oferta: -1 pendiente,
             // rechazada, 1 aprobada.
} Req;
Subasta nuevaSubasta(int n) {
  Subasta s= malloc(sizeof(*s));
  s->n= n;
  s->c= MakePriQueue();
  return s;
}
int ofrecer(Subasta s, int oferta) {
  Req r= {oferta, -1};
  PriPut(s->c, &r, oferta);
  if (PriSize(s->c)>s->n) {
    Req *pr= PriGet(s->c);
    pr->resol= 0; // se rechaza
  }
  while (r.resol == -1)
    ;
  return r.resol;
}
int adjudicar(Subasta s, int *punid) {
  int recaud= 0;
  *punid= 0;
  while (!EmptyPriQueue(s->c)) {
    Req *pr= PriGet(s->c);
    pr->resol= 1; // se adjudica
    (*punid)++;
    recaud += pr->oferta;
  }
  s->n -= *punid;
  return recaud;
}
```

Este código por sí solo debe ser suficiente para que Ud. comprenda qué es lo que debe hacer el sistema de subastas. Estúdielo detenidamente y descubrirá lo siguiente. La implementación funciona correctamente si las operaciones no se invocan simultáneamente. Un propietario crea su subasta invocando *nuevaSubasta*. En *n* indica el número de unidades idénticas del producto que subastará. Múltiples cores invocan en paralelo *ofrecer* para hacer una oferta por una unidad de la subasta *s*. El dinero ofrecido se indica en *oferta*. La función *ofrecer* espera hasta que se resuelva si se adjudicó o no la unidad. Esto ocurre cuando se llega a los *n* cores que ya hicieron una oferta mayor (por simplicidad suponga que todas las ofertas son distintas) en cuyo caso se retorna 0; o cuando el propietario llama a *adjudicar* y en ese caso *ofrecer* retorna 1. La función *adjudicar* retorna el total recaudado y entrega el número de unidades que se subastaron en **punid* (menor que *n* si no hay suficientes oferentes).

Observe que cuando llega un oferente se usa *PriPut* para encolar un objeto *r* de tipo *Req* en la cola *c* de la subasta. Su prioridad es la oferta. Si se llega a *n+1* oferentes, se extrae con *PriGet* la oferta menor y se rechaza.

Reprograme el código anterior de manera correcta y eficiente. Los spin-locks son la única herramienta disponible para sincronizar los distintos cores. Para hacer esperar un core Ud. debe usar un spin-lock. No se permiten otras formas de busy-waiting. Ud. sí dispone de *malloc*, del tipo *PriQueue* y sus operaciones.

Pregunta 2

Implemente la siguiente API de forma nativa en nSystem:

```
nSubasta nNuevaSubasta(int n);
Resol nOfrecer(nSubasta s, double oferta);
double nAdjudicar(nSubasta s, int *punid);
```

Debe reimplementar la misma funcionalidad de la pregunta 1 usando las funciones de bajo nivel de nSystem (*START_CRITICAL*, *Resume*, *PutTask*, etc.). Ud. no puede hacer busy-waiting. Sí dispone del tipo *PriQueue*. Debe ser eficiente evitando cambios de contexto innecesarios.

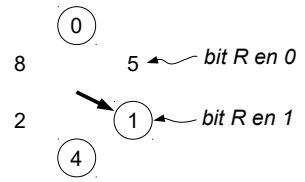
Pregunta 3

a.- 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?

b.- Explique qué estrategia de scheduling usaría si necesita entregar: (i) un buen tiempo de respuesta, (ii) un buen tiempo de despacho.

c.- Explique por qué no tiene sentido implementar la estrategia del working set en un sistema monoproceso. ¿Tendría sentido implementar la estrategia del reloj?

d.- Considere un sistema Unix que implementa la estrategia del reloj. El sistema posee 6 páginas reales disponibles y corre un solo proceso. La figura de la derecha 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 6 0 1 7 8.



e.- Los primeros computadores personales no disponían de una MMU (*memory management unit*). Explique por qué esto era incómodo para un desarrollador que editaba su programa al mismo tiempo que lo compilaba y ejecutaba. ¿Cuáles eran las consecuencias cuando su programa tenía un error de manejo de punteros, comparado con los computadores modernos que sí tienen una MMU?