Pregunta 1

Múltiples threads productores de oxígeno e hidrógeno deben formar moléculas de agua (H_2O):

```
void productorOxi() {
  for (;;) {
    Oxigen *o= ...;
    H2O *agua= combinarOxi(o);
    energia(agua);
} }
```

```
void productorHidro() {
  for (;;) {
    Hidrogeno *h= ...;
    H2O *agua= combinarHidro(h);
    condensar(agua);
} }
```

Por supuesto se deben combinar 2 átomos de hidrógeno con 1 átomo de oxígeno para formar H_2O. La siguiente es una implementación incorrecta de *combinarOxi* y *combinarHidro*, pero funciona cuando no hay invocaciones simultáneas (note que *barrera* parte con 1 ticket y los otros 2 semáforos con 0 tickets):

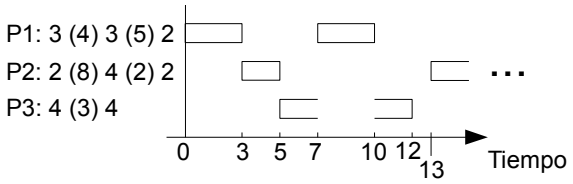
```
Hidrogeno *h1= NULL, *h2= NULL;
H2O *h2o;
nSem barrera; /* 1 ticket */
nSem sem_h2o; /* 0 tickets */
nSem sem_h;   /* 0 tickets */
H2O *combinarOxi(Oxigeno *o) {
  nWaitSem(barrera);
  nWaitSem(sem_h);
  nWaitSem(sem_h);
  h2o= makeH2O(o, h1, h2);
  h1= h2= NULL;
  nSignalSem(sem_h2o);
  nSignalSem(sem_h2o);
  nSignalSem(barrera);
  return h2o;
}
```

```
H2O *combinarHidro(Hidrogeno *h)
{
  while (h1!=NULL && h2!=NULL)
    ;
  if (h1==NULL)
    h1= h;
  else
    h2= h;
  nSignalSem(sem_h);
  nWaitSem(sem_h2o);
  H2O *res= h2o;
  return res;
}
```

Parte a.- (1 punto) Haga un diagrama de threads que muestre que bastan 2 invocaciones simultáneas de *combinarHidro* para que se pierda un átomo de hidrógeno.

Parte b.- (3 puntos) Corrija esta solución manteniendo su espíritu. Es decir haga pequeñas modificaciones. Para ello use 2 semáforos adicionales. Asegúrese de que su solución no sufre del mismo problema de la parte a.-

Parte c.- El diagrama de arriba a la derecha muestra el scheduling de 3 procesos. En tiempo 0 los 3 procesos están READY. 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: I. Ordene los procesos de mejor a peor prioridad (0,5 puntos) II. Explique si se trata de scheduling *preemptive* o *non-preemptive* (0,5 puntos) III. Complete el diagrama (1 punto).



Pregunta 2

Un sistema *batch* permite solicitar la ejecución de un *job* por medio de la función *submitJob* y esperar hasta que ese job termine con *waitJob*. La siguiente implementación es correcta pero ineficiente:

```
typedef struct {
  Code *code;
  Input *in;
} Request;

Request *submitJob(
    Code *code, Input *in) {
  Request *req=
        nMalloc(sizeof(*req));
  req->code= code;
  req->in= in;
  return req;
}
```

```
Monitor mon;

void initBatch() {
  mon= nMakeMonitor();
}

Output *waitJob(Request *req) {
  nEnter(mon);
  Output *out= execute(
         req->code, req->in, 0);
  nExit(mon);
  return out;
}
```

Reprograme esta solución de tal forma que los jobs comiencen a ejecutarse inmediatamente después de la invocación de *submitJob*, en la medida que hayan cores disponibles. El sistema cuenta con 8 cores. Ud. especifica el número del core en que se debe ejecutar un job por medio del 3er parámetro de *execute*.

Restricciones: Ud. debe aprovechar de la mejor manera los 8 cores disponibles. Un core puede ejecutar un solo job a la vez. Para la sincronización Ud. debe usar un solo monitor.

Concretamente se pide reeimplementar *Request*, *initBatch*, *submitJob* y *waitJob*. Encole las solicitudes en una cola fifo. Use 8 threads adicionales para ejecutar los jobs, uno por cada core.

API para manejo de colas FIFO:

```
FifoQueue MakeFifoQueue();
int EmptyFifoQueue(FifoQueue q);
void PutObj(FifoQueue q, void *obj);
void *GetObj(FifoQueue q);
```