Pregunta 1

```
double f(double x) {
  return p(x) ? g(x) : h(x);
}
```

Parte a.- En la función *f* del cuadro de la derecha, *p(x)*, *g(x)* y *h(x)* toman mucho tiempo en calcularse. Se sabe que *p(x)* es verdadero en el 99% de los casos. **Modifique** *f* de manera que se calcule *p(x)* en paralelo con *g(x)*, especulando que *p* será verdadero. Si *p* es verdadero, entregue el resultado de *g(x)*, y si no tome un tiempo adicional para calcular y entregar *h(x)*. No puede crear más de un thread adicional.

Parte b.- Para solicitar un recurso único compartido, *N* threads invocan *pedir(id)*, en donde *id* está entre *[0,N-1]* e identifica al thread solicitante, e invocan *devolver(id)*, para notificar que lo desocuparon, siempre con el mismo *id* con el que lo solicitaron. Entre los threads en espera del recurso, se debe otorgar a aquel que lo obtuvo por última vez hace más tiempo. La siguiente implementación funciona correctamente el 99,9% de los casos.

```
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
int ocup= 0;
PriQueue *q; // = makePriQueue( );
int pri[N];  // inician en 0
```

```
typedef struct {
  int ready;
  pthread_cond_t w;
} Request;
```

```
void pedir(int id) {
  pthread_mutex_lock(&m);
  if (ocup) {
    Request req= {0,PTHREAD_COND_INITIALIZER};
    priPut(q, &req, pri[id]);
    while (!req.ready)
      pthread_cond_wait(&req.w,&m);
  }
  ocup= 1;
  pri[id]= getTime();
  pthread_mutex_unlock(&m);
}
```

```
void devolver(int id) {
  pthread_mutex_lock(&m);
  ocup= 0;
  if (!priEmpty(q)) {
    Request *preq=
       priGet(q);
    preq->ready= 1;
    pthread_cond_signal(
                &preq->w);
  }
  pthread_mutex_unlock(&m);
}
```

Haga un diagrama de threads que muestre que 2 threads podrían llegar a ocupar el recurso simultáneamente y por lo tanto la implementación es incorrecta. *Ayuda*: Considere el caso en que 2 threads esperan obtener el mutex, uno espera en *pthread_mutex_lock* y el otro en *pthread_cond_wait*.

Parte c.- **Corrija** esta implementación. Recuerde que debe otorgar el recurso a aquel thread en espera que obtuvo el recurso por última vez hace más tiempo.

Pregunta 2

I. La siguiente es una implementación parcial del mismo problema de la pregunta 1.c, pero que usa una sola condición. **Programe** la función *pedir* sin modificar el código de *devolver*.

```
int ocup= 0;
int pri[N]; // inician en 0
pthread_mutex_t m=
          PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c=
          PTHREAD_COND_INITIALIZER;
PriQueue *q; // = makePriQueue( )
```

```
void devolver(int id) {
  lock(&m);
  ocup= 0;
  broadcast(&c);
  unlock(&m);
}
void pedir(int id) { ... }
```

Le será de utilidad *void *priPeek(PriQueue *q)* que entrega el elemento de mejor prioridad (como *priGet* pero sin extraerlo).

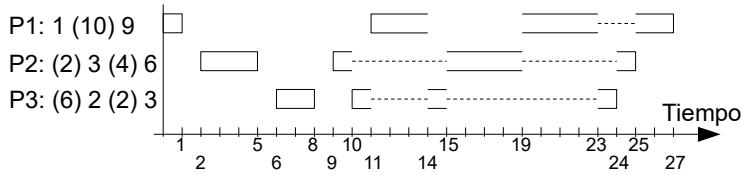
II. Implemente una solución nativa para nThreads del mismo problema de la pregunta 1.c. Las funciones que debe **programar** son:

```
void nPedir( );          // Solicitud de uso del recurso
void nDevolver( );       // Notificación de devolución
void nth_iniRecurso( );  // Para inicializar variables globales
```

Use el campo *pri* en el descriptor de thread para almacenar la prioridad.

Restricción: No puede usar herramientas de sincronización preexistentes en nThreads como mutex, condiciones o semáforos. Debe usar operaciones como *START_CRITICAL*, *END_CRITICAL*, *schedule*, *setReady*, *nth_putBack*, *nth_peekFront* (obtiene el primer thread de una NthQueue sin extraerlo), etc. Puede usar *PriQueue*.

III. El diagrama muestra el scheduling *round robin* de 3 procesos:



Junto a cada proceso se indica la duración de las ráfagas de CPU y entre paréntesis la duración de los estados de espera. La línea punteada indica cuando el proceso esta READY y el espacio en blanco que está en estado de espera. En tiempo 10 el scheduler decide ejecutar P3 quedando en la cola P2; en 11 decide ejecutar P1, quedando en la cola (P3, P2) en ese orden; en 14 decide ejecutar P3, quedando en la cola (P2, P1).

Responda: (a) Para los siguientes instantes en el que scheduler decida entre 2 o más procesos, indique el proceso que decide ejecutar y el contenido de la cola. (b) Rehaga el diagrama cuando el scheduling es SJF non preemptive, considerando como estimador la duración de la última ráfaga.