Apellidos: **Nombre:**

Concurrencia y Paralelismo

Parte I: Concurrencia

Examen Julio 2012

1. **Algorítmo de los barberos** [2.5 puntos]

   Partimos del ejemplo dado en clase de tener múltiples barberos. Existe una cola para que los clientes se atiendan en orden de llegada. Cambiar el código para que un cliente pueda pedir esperar por un barbero en particular `barber` que se le pase como parámetro. Debe haber una cola por cada barbero, y una para todos los barberos (los clientes que no les importa el barbero la llaman con `-1` como número de barbero).

```
struct queue customers;
struct queue barbers[NUMBARBERS];
struct customer {
  pthread_cond_t wait;
  int num;
}
bool queue_is_empty(&struct queue queue);
void insert_customer(&struct queue queue, struct customer *customer);
struct customer *retrieve_customer(&struct queue queue);
int waiting_customers;

void *barber_function(void *ptr)
{
        struct barber_info *t = ptr;

        while(true) {
                struct customer *customer;
                pthread_mutex_lock(&mutex);

                while (queue_is_empty(&customers)) {
                        pthread_cond_wait(&no_customers, &mutex);
                }

                customer = retrieve_customer(customers);

                pthread_cond_signal(&customer->wait);

                cut_hair(t->barber_num, customer->num);
                free(customer);
                }
        }
}

void *customer_function(void *ptr)
{
        struct customer_info *t = ptr;
        struct customer *customer = malloc(sizeof(struct customer));
        int barber = t->barber_num;

        pthread_mutex_lock(&mutex);
        if(waiting_customers == num_chairs)  {
                printf("waiting_room_full_for_customer_%d\n", t->customer_num);
                pthread_mutex_unlock(&mutex);
                return NULL;
        }

        pthread_cond_init(&customer->wait, NULL);
        customer->num = t->customer_num;

        insert_customer(customers, customer);

        pthread_cond_broadcast(&no_customers);
        waiting_customers++;
        pthread_cond_wait(&customer->wait, &mutex);
        waiting_customers--;
        get_hair_cut(t->customer_num);
}
```

2. **Sistema Caché** [2.5 pts]

En un sistema hay un dispositivo de almacenamiento lento pero de mucha capacidad. Para acelerar las operaciones contra este sistema se desea desarrollar un sistema de caché en memoria que guarde los bloques más recientemente usados, y permita a los distintos procesos que quieran operar contra ese dispositivo leer y escribir simultáneamente de esa caché. La caché se implementará con un array de bloques `char *cblock[CACHE_SIZE]`, donde podrán almacenarse hasta `CACHE_SIZE` bloques.

Como la capacidad de la memoria es más pequeña que la del dispositivo solo podremos tener una parte de los bloques del dispositivo en memoria en un determinado momento. Para decidir donde se guarda cada bloque que se use en la caché se utiliza una función de hash que a partir del número de bloque nos dice la posición en `cblock` dondé estará. (`int hash(int bnum)`). Para saber que bloque está almacenado actualmente en cada posición del array cblock se usa el array `int block_num[CACHE_SIZE]`, de tal forma que `block_num[i]` contiene el número de bloque almacenado en `cblock[i]`.

Para mejorar la eficiencia los bloques que se escriban se mantendrán con las modificaciones en memoria hasta que haya que sustituirlos por otro bloque. Para saber si han cambiado o no, y por tanto si hay que escribir los cambios al dispositivo o no, se usa el array `int dirty[CACHE_SIZE]` que indica si un bloque tiene cambios con respecto a lo almacenado en el dispositivo. Cada vez que se escriba un bloque habrá que marcarlo como usado.

La información del estado actual del sistema caché se guarda en los siguientes campos:

```
char *cblock[CACHE_SIZE];
int block_num[CACHE_SIZE];
int num_readers[CACHE_SIZE];
int dirty[CACHE_SIZE];
pthread_mutex_t *lock;
pthread_mutext_t *block_lock[CACHE_SIZE];
pthread_cond_t *waiting_block[CACHE_SIZE];
```

Partiendo de las siguientes funciones de lectura y escritura añada los bloqueos necesarios para que no se produzcan problemas por el acceso concurrente de varios procesos al sistema caché. Intente minimizar el tiempo que se bloquea el acceso al sistema caché.

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex); // No duerme nunca al thread, devuelve 0 si bloquea
                                                   // y el error EBUSY si ya esta bloqueado.
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int read(int bnum) {
        int hash_id = hash(bnum);

        if(block_num[hash_id] != bnum) {
                if(dirty[hash_id]==TRUE) write_block_to_device(block_num[hash_id], cblock[hash_id]);
                read_block_from_device(bnum, cblock[hash_id]);
                block_num[hash_id] = bnum;
                dirty[hash_id] = FALSE;
        }

        // Hacer algo con cblock[hash_id]
}

int write(int bnum, char *block) {
        int hash_id = hash(bnum);

        if(block_num[hash_id] != bnum && dirty[hash_id] == TRUE)
                write_block_to_device(block_num[hash_id], cblock[hash_id]);

        memcpy(cblock[hash_id], block, BLOCK_SIZE);
        block_num[hash_id] = bnum;
        dirty[hash_id] = TRUE;
}
```