Semaphores implemented using pthreads

```c
/* File: bksem.h */
* Author: Keller
* Purpose: Bob Keller's semaphores implemented using pthread primitives
* Declare semaphore as:
* struct bksem s;
* int init(&s, value);
* where value should be non-negative.
* Operation up, signal, or V:
* up(&s);
* Operation down, wait, or P:
* down(&s)
*/

#define _REENTRANT
#include <pthread.h>

typedef struct
{
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int value;
} bksem;

/* workings:
* The value of the semaphore maintains the following invariant, assuming
* that the initial value is non-negative, which it always should be:
* If the value is <= 0, this is the number of threads waiting on the
* semaphore (i.e. on the associated condition variable).
* If the value is <= 0, this is the number of times threads can perform
* the down operation without waiting.
*/

void init(bksem* s, int value)
{
    s->value = value;
}

void up(bksem* s)
{
    pthread_mutex_lock(&(s->mutex));
    s->value++;
    if ( s->value <= 0 )
    {
        pthread_cond_signal(&(s->cond));
    }
    pthread_mutex_unlock(&(s->mutex));
}

void down(bksem* s)
{
    pthread_mutex_lock(&(s->mutex));
    s->value--;
    if ( s->value < 0 )
    {
        pthread_cond_wait(&(s->cond), &(s->mutex));
    }
    pthread_mutex_unlock(&(s->mutex));
}
```

Test Program

```c
/* struct representing shared data */
typedef struct
{
    int consumerDelay;
    int producerDelay;
    bksem supply;
    bksem space;
    bksem mutex;
    int occupied;
    int vacant;
} sharedData;

sharedData pkg;

void* producer(void* arg)
{
    int i;
    for( i = 0; i < CYCLES; i++ )
    {
        sleep(pkg.producerDelay); // producer delay
        down(&pkg.space); // wait for space
        down(&pkg.mutex); // lock data
        pkg.occupied++;
        // simulate production
        printf("Producer produces %d, occupied = %d, vacant = %d
", i, pkg.occupied, pkg.vacant);
        up(&pkg.mutex); // unlock data
        up(&pkg.supply); // indicate production
    }
}

void* consumer(void* arg)
{
    int i;
    for( i = 0; i < CYCLES; i++ )
    {
        down(&pkg.supply); // wait for supply
        sleep(pkg.consumerDelay); // consumer delay
        down(&pkg.mutex); // lock data
        pkg.occupied--;
        // simulate consumption
        printf("Consumer consumes cycle %d, occupied = %d, vacant = %d
", i, pkg.occupied, pkg.vacant);
        up(&pkg.mutex); // unlock data
        up(&pkg.space); // indicate consumption
    }
}
<table>
<thead>
<tr>
<th>Exercise</th>
<th>Petri Net modeling</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Implement a barrier synchronization mechanism for Posix threads.</td>
<td>● <em>Gracefully</em> models: mutexes semaphores</td>
</tr>
<tr>
<td></td>
<td>● Not so great at broadcast unless number of recipients is fixed.</td>
</tr>
</tbody>
</table>