#include <stdio.h>

#include <assert.h>

#include<string.h>

#include <sys/time.h>

#include <pthread.h>

int NumCats,NumMice,Numbowls,cats\_eats,mouse\_eats;

#define NumBowls 2 /\* number of Bowls \*/

const int cat\_wait=10; //global constant declaration of waiting time of cat to eat

const int cat\_eat=4;

const int mouse\_wait=10; //global constant declaration of waiting time of cat to eat

const int mouse\_eat=2;

typedef struct FoodBowl

{

int free\_Bowls;

int cats\_eating;

int mice\_eating;

int cats\_waiting;

enum {

none\_eating,

cat\_eating,

mouse\_eating

} status[NumBowls];

pthread\_mutex\_t mutex;

pthread\_cond\_t free\_cv;

pthread\_cond\_t cat\_cv;

} FoodBowl\_t;

int NumCats,NumMice;

static const char \*progname = "CAT AND MICE SYNCHRONIZATION PROBLEM";

static void Display(const char \*name,const char \*what,FoodBowl\_t \*FoodBowl, int my\_FoodBowl)

{

int i;

printf("\n[");

for (i = 0; i < NumBowls; i++) {

if (i)

printf(":");

switch (FoodBowl->status[i]) {

case none\_eating:

printf("-");

break;

case cat\_eating:

printf("cat");

break;

case mouse\_eating:

printf("mice");

break;

}

}

printf("] %s %s eating from FoodBowl %d\n", name, what, my\_FoodBowl);

}

/\*

According to question

Cats wait for a Bowl to become free and when coming to the Bowl,

they wait for the mice to run away from the Bowl (they are nice

cats and want to eat FoodBowl from Bowl rather than mice). Eating is simulated

by a simple sleep().

\*/

static void\*cat(void \*arg)

{

FoodBowl\_t \*FoodBowl = (FoodBowl\_t \*) arg;

int n = cats\_eats;

int my\_FoodBowl = -1;

int i;

for (n = cats\_eats; n > 0; n--) {

pthread\_mutex\_lock(&FoodBowl->mutex);

pthread\_cond\_broadcast(&FoodBowl->cat\_cv);

FoodBowl->cats\_waiting++;

while (FoodBowl->free\_Bowls <= 0 || FoodBowl->mice\_eating > 0)

{

pthread\_cond\_wait(&FoodBowl->free\_cv, &FoodBowl->mutex);

}

FoodBowl->cats\_waiting--;

assert(FoodBowl->free\_Bowls > 0);

FoodBowl->free\_Bowls--;

assert(FoodBowl->cats\_eating < NumCats);

FoodBowl->cats\_eating++;

for (i = 0; i < NumBowls && FoodBowl->status[i] != none\_eating; i++) ;

my\_FoodBowl = i;

assert(FoodBowl->status[my\_FoodBowl] == none\_eating);

FoodBowl->status[my\_FoodBowl] = cat\_eating;

Display("cat", "started", FoodBowl, my\_FoodBowl);

pthread\_mutex\_unlock(&FoodBowl->mutex);

sleep(cat\_eat);

pthread\_mutex\_lock(&FoodBowl->mutex);

assert(FoodBowl->free\_Bowls < NumBowls);

FoodBowl->free\_Bowls++;

assert(FoodBowl->cats\_eating > 0);

FoodBowl->cats\_eating--;

FoodBowl->status[my\_FoodBowl] = none\_eating;

pthread\_cond\_broadcast(&FoodBowl->free\_cv);

Display("cat","finished", FoodBowl, my\_FoodBowl);

pthread\_mutex\_unlock(&FoodBowl->mutex);

sleep(rand() % cat\_wait);

}

return NULL;

}

/\*

\* Mice wait for a FoodBowl to become free and cats go away. While eating,

\* they have to check that no cat is coming. Hence, a simple sleep()

\* cannot be used.

\*/

static void\*mouse(void \*arg)

{

FoodBowl\_t \*FoodBowl = (FoodBowl\_t \*) arg;

int n = mouse\_eats;

struct timespec t1;

struct timeval t2;

int my\_FoodBowl;

int i;

for (n = mouse\_eats; n > 0; n--)

{

pthread\_mutex\_lock(&FoodBowl->mutex);

while (FoodBowl->free\_Bowls <= 0 || FoodBowl->cats\_eating > 0 || FoodBowl->cats\_waiting > 0)

{

pthread\_cond\_wait(&FoodBowl->free\_cv, &FoodBowl->mutex);

}

assert(FoodBowl->free\_Bowls > 0);

FoodBowl->free\_Bowls--;

assert(FoodBowl->cats\_eating == 0);

assert(FoodBowl->mice\_eating < NumMice);

FoodBowl->mice\_eating++;

for (i = 0; i < NumBowls && FoodBowl->status[i] != none\_eating; i++) ;

my\_FoodBowl = i;

assert(FoodBowl->status[my\_FoodBowl] == none\_eating);

FoodBowl->status[my\_FoodBowl] = mouse\_eating;

Display("mouse","started", FoodBowl, my\_FoodBowl);

pthread\_mutex\_unlock(&FoodBowl->mutex);

t1.tv\_sec = t2.tv\_sec;

t1.tv\_nsec = t2.tv\_usec \* 1000;

t1.tv\_sec += mouse\_eat;

pthread\_mutex\_lock(&FoodBowl->mutex);

pthread\_cond\_timedwait(&FoodBowl->cat\_cv, &FoodBowl->mutex, &t1);

pthread\_mutex\_unlock(&FoodBowl->mutex);

pthread\_mutex\_lock(&FoodBowl->mutex);

assert(FoodBowl->free\_Bowls < NumBowls);

FoodBowl->free\_Bowls++;

assert(FoodBowl->cats\_eating == 0);

assert(FoodBowl->mice\_eating > 0);

FoodBowl->mice\_eating--;

FoodBowl->status[my\_FoodBowl]=none\_eating;

pthread\_cond\_broadcast(&FoodBowl->free\_cv);

Display("mouse","finished", FoodBowl, my\_FoodBowl);

pthread\_mutex\_unlock(&FoodBowl->mutex);

/\* sleep to avoid Starvation \*/

sleep(rand() % mouse\_wait);

}

return NULL;

}

int main(int argc, char \*argv[])

{

int i, err;

FoodBowl\_t \_FoodBowl, \*FoodBowl;

printf("Enter the number of cats: ");

scanf("%d",&NumCats);

printf("Enter the number of Mice: ");

scanf("%d",&NumMice);

printf(" how many times a cat wants to eat: ");

scanf("%d",&cats\_eats);

printf(" how many times a mice wants to eat: ");

scanf("%d",&mouse\_eats);

pthread\_t cats[NumCats];

pthread\_t mice[NumMice];

srand(time(NULL));

FoodBowl = &\_FoodBowl;

memset(FoodBowl, 0, sizeof(FoodBowl\_t));

FoodBowl->free\_Bowls = NumBowls;

pthread\_mutex\_init(&FoodBowl->mutex, NULL);

pthread\_cond\_init(&FoodBowl->free\_cv, NULL);

pthread\_cond\_init(&FoodBowl->cat\_cv, NULL);

printf("\n !!!Cat And Mice Synchronization for Eating from Bowls!!!\n\n");

printf("[B1,B2] BOWLS OCCUPIED BY ");

for (i = 0; i < NumCats; i++) {

err = pthread\_create(&cats[i], NULL, cat, FoodBowl);

if (err != 0) {

printf("unable to create cat thread");

}

}

for (i = 0; i < NumMice; i++) {

err = pthread\_create(&mice[i], NULL, mouse, FoodBowl);

if (err != 0)

{

printf(" unable to create mouse thread");

}

}

for (i = 0; i < NumCats; i++)

{

pthread\_join(cats[i], NULL);

}

for (i = 0; i < NumMice; i++)

{

pthread\_join(mice[i], NULL);

}

pthread\_mutex\_destroy(&FoodBowl->mutex);

pthread\_cond\_destroy(&FoodBowl->free\_cv);

pthread\_cond\_destroy(&FoodBowl->cat\_cv);

return 0;

}