```c
#include "_kernelCore.h"
#include "osDefs.h"

//global variables
extern threadStruct threadCollection[MAX_THREADS];
mutexStruct mutexCollection[MAX_THREADS];
extern int numThreads;
int threadCurr = 0;
int numMutex = 0;
extern int idleIndex;
bool leaveIdle = false;


//set priority of the PendSV interrupt
void kernelInit(void){
  //PendSV priority
  SHPR3 |= 0xFE << 16;
  SHPR3 |= 0xFFU << 24; //SysTick priority

  SHPR2 |= 0xFDU << 24; //SVC priority
}

//start running the kernel, i.e. the OS
bool osKernelStart(){
  threadCurr = 0;
  if(numThreads > 0)
  {
    __set_CONTROL(1<<1); //enter threading
    __set_PSP((uint32_t) threadCollection[threadCurr].TSP); //set PSP to the first thread address

    osLoadFirst(); //begin running threads
  }

  return false; //once called, function should not end unless something went wrong in OS
}

//create a new mutex in the mutex struct array
void osCreateMutex(){
  mutexCollection[numMutex].available = true;
  mutexCollection[numMutex].currentOwner = NONE;

  numMutex++;
}

//determine if the thread is allowed to run otherwise block
void osAcquireMutex(int mutexID){
  //if running thread can acquire the mutex, then acquire and proceed running thread
  if (mutexCollection[mutexID].available == true){
    mutexCollection[mutexID].available = false;
    mutexCollection[mutexID].currentOwner = threadCurr;

    printf("Mutex %d acquired by Thread %d\n", mutexID, threadCurr+1);
  }
  //if the mutex is already claimed by another thread, add this thread to the blocked list and switch it out
  else if(mutexCollection[mutexID].available == false && mutexCollection[mutexID].currentOwner != threadCurr){
    threadCollection[threadCurr].status = BLOCKED;
    threadCollection[threadCurr].timer = 0; //start counting time blocked
    threadCollection[threadCurr].waitMutex = mutexID;

    printf("Mutex %d already claimed by Thread %d. Set Thread %d to blocked\n", mutexID, mutexCollection[mutexID].currentOwner+1, threadCurr+1);

    //switch out thread, by a mutex yield
    __ASM("SVC #1");
  }
  else{
    printf("Repeated claim of mutex %d\n", mutexID);
  }
}

```

```c
//make mutex available and/or give it to the next blocked thread
void osReleaseMutex(int mutexID){
  int longestWait = 0;
  bool isFound = false;

  //do not do release mutex sequence if called by a thread that is not the mutex owner
  if (mutexCollection[mutexID].currentOwner == threadCurr){

    for (int i = 0; i < numThreads; i++){
      //look for at least one thread blocked by the current mutex
      if (threadCollection[i].status == BLOCKED && threadCollection[i].waitMutex == mutexID){
        isFound = true;
        longestWait = i;

        printf("Found Thread %d. Timer: %d\n", i+1, threadCollection[i].timer);
      }

      //if found, start looking for the thread that has been waiting the longest
      if (isFound == true && threadCollection[i].timer < threadCollection[longestWait].timer){
        longestWait = i;

        printf("Found Thread %d. Timer: %d\n", i+1, threadCollection[i].timer);
      }
    }
    if (isFound == true){
      //remove longestWait thread from the blocked queue, so that it is in the round-robin next time scheduler() is called
      threadCollection[longestWait].status = WAITING;
      threadCollection[longestWait].timer = threadCollection[longestWait].timeslice;
      threadCollection[longestWait].waitMutex = NONE;

      //give longestWait thread the mutex
      mutexCollection[mutexID].available = false;
      mutexCollection[mutexID].currentOwner = longestWait;

      printf("Give Mutex %d to Thread %d\n", mutexID, longestWait+1);
    }
    else{ //if no blocked threads, just make the mutex available
      mutexCollection[mutexID].available = true;
      mutexCollection[mutexID].currentOwner = NONE;

      printf("No blocked threads for Mutex %d, is now available\n", mutexID);
    }
  }

  else{
    printf("Thread %d does not own mutex %d. Will not release.\n", threadCurr, mutexID);
  }
}


//start running the first thread, which will lead into context switching between all the threads
void osLoadFirst(){
    ICSR |= 1<<28;
    __asm("isb");
}

//called when a thread yields, starts task switching process
void osYield(void){
  //Call SVC
  __ASM("SVC #0");
}

//determine next available thread to switch to
void scheduler(void){
  bool isFound = false;
  int index = threadCurr;
  if(threadCurr == idleIndex){
    index = threadCurr-1; //want to cycle through the valid thread indexes
  }

  if (numThreads > 1){
    for (int i = 0; i < numThreads && isFound == false; i++){
      //cycle through the threads in the thread struct array
      index = (index+1)%numThreads;
      printf("Trying thread %d\n", index+1);

      //check status: if next thread in round robin is waiting, proceed! else, loop back and look at next thread
      if(threadCollection[index].status == WAITING){
        threadCurr = index;
        isFound = true;
      }
    }

    //if no threads are waiting, use idle thread
    if(isFound != true){
      printf("all the people of the world are asleep\n");
      threadCurr = idleIndex;
    }
  }
}

void SysTick_Handler(void){
  //printf("thread num %d, timer: %d\n", threadCurr+1, threadCollection[threadCurr].timer);

  //decrement the running thread's timeslice, sleep timers, and blocked timers
  for(int i = 0; i < numThreads; i++){
    if(threadCollection[i].status != WAITING){
      --threadCollection[threadCurr].timer;
    }

    if(threadCollection[i].status == SLEEPING)
    {
      /*if(threadCollection[i].timer % 50 == 0){
        printf("Thread %d sleeptime: %d \n", (i+1), threadCollection[i].timer);
      }*/

      //check wake-up status
      if(threadCollection[i].timer <= 0)
      {
        if(threadCurr == idleIndex){
          leaveIdle = true;
        }
        threadCollection[i].status = WAITING;
        threadCollection[i].timer = threadCollection[i].timeslice;
      }
    }
  }

  //if timeslice of running thread is up, proceed with task-switching
  if((threadCollection[threadCurr].timer <= 0 && threadCollection[threadCurr].status != BLOCKED) || leaveIdle == true)
  {
    printf("Thread %d timer complete\n", threadCurr+1);
    threadCollection[threadCurr].TSP = (uint32_t*)(__get_PSP()-8*4); //decrement PSP only 8 locations lower, since the hardware registers remain on the stack
    //prepare current thread to sleep if can sleep
    if(threadCollection[threadCurr].sleepTime != 0){
      threadCollection[threadCurr].status = SLEEPING;
      threadCollection[threadCurr].timer = threadCollection[threadCurr].sleepTime;
    }
    //if thread doesn't sleep, set status to waiting and timer to timeslice
    else{
      threadCollection[threadCurr].status = WAITING;
      threadCollection[threadCurr].timer = threadCollection[threadCurr].timeslice;
    }
    //if a thread woke up, return to round robin
    if(leaveIdle == true){
      threadCurr = idleIndex-1;
      leaveIdle = false;
    }

    scheduler();

    ICSR |= 1<<28;
    __asm("isb");
  }
}

void SVC_Handler_Main(uint32_t *svc_args)
{
  char call = ((char*)svc_args[6])[-2];

  if(call == 0) //from osYield()
  {
    //move TSP of the running thread 16 memory locations lower, so that next time the thread loads the 16 context registers, we end at the same PSP
    threadCollection[threadCurr].TSP = (uint32_t*)(__get_PSP()-8*4);

    //if the thread is able to sleep, set it to sleep
    if(threadCollection[threadCurr].sleepTime != 0){
      threadCollection[threadCurr].status = SLEEPING;
      threadCollection[threadCurr].timer = threadCollection[threadCurr].sleepTime; //set timer to user-defined sleep timer
    }
    //otherwise, set it back to waiting
    else{
      threadCollection[threadCurr].status = WAITING;
    }

    scheduler();

    ICSR |= 1<<28;
    __asm("isb");
  }

  if(call == 1){ //from thread being blocked in osAcquireMutex()
    threadCollection[threadCurr].TSP = (uint32_t*)(__get_PSP()-8*4);
    scheduler();

    ICSR |= 1<<28;
    __asm("isb");
  }
}


int task_switch(void){
  //set PSP to the thread we want to start running
  __set_PSP((uint32_t)threadCollection[threadCurr].TSP);
  threadCollection[threadCurr].status = ACTIVE;

  if (threadCurr == numThreads ){
    printf("Running idle thread \n");
  }
  return 0;
}
```