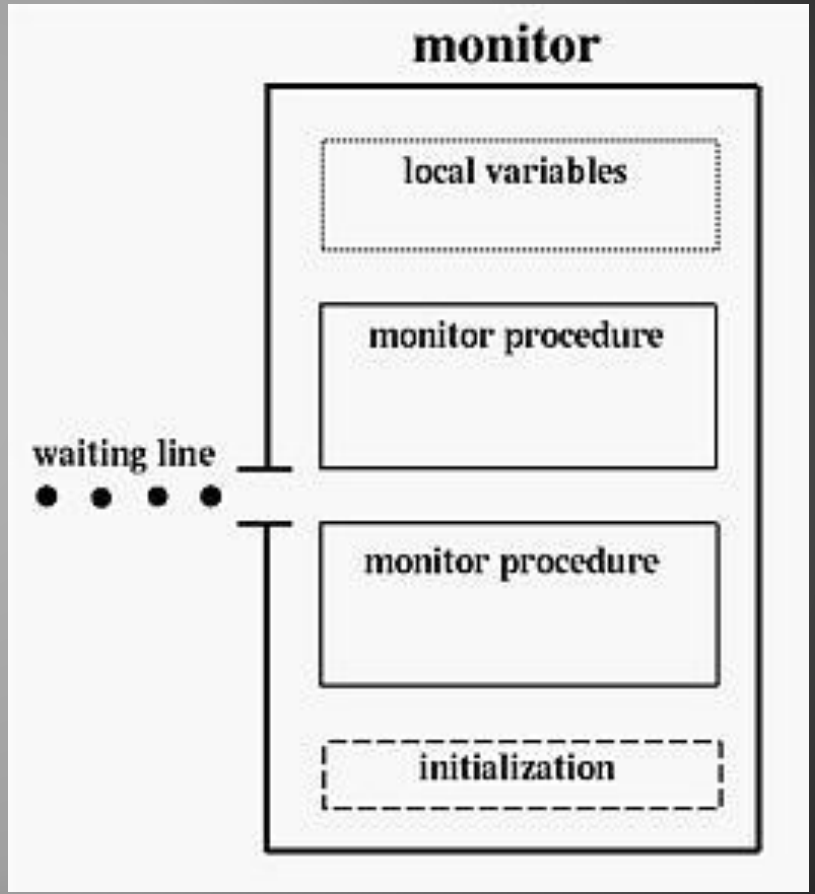
Threads in C
Implementing Monitors

EPL222 – Lab6

Simulating a Monitor

- Recall that a monitor has some **local variables**, monitor **procedures** and an **initialization** part
- When the monitor is created, the initialization part initializes the local variables
- The boundary of the monitor serves as a fence for blocking threads that call one of the monitor procedures
  - More precisely, at any time, there can only be **one thread executing within the monitor** boundary
  - Therefore, the monitor boundary guarantees mutual exclusion



Simulating a Monitor

- To simulate a monitor using C, we need to address three questions:
  - (1) how do we make sure that the local variables will not be accessed by non-monitor procedures
  - (2) how do we make sure that the user can only "see" the interface without knowing the details of the monitor
  - (3) how do we properly setup the monitor boundary so that mutual exclusion can be guaranteed
- Questions (1) and (2) have a simple solution
  - We could use two files, one header file that stores the prototypes of the monitor procedures and a implementation file in which the local variables and all monitor procedures that are not supposed to be used by the user directly are declared with **static** storage class
    - A name declared with static can only be accessed from the place after the declaration point in the "same"' file
- Question (3) has a natural answer: use a mutex lock!

Counter Monitor

- We shall design a monitor for maintaining a counter so that its value can be increased, decreased, reset and retrieved
  - In addition to this, we also need an initialization function
- We need a header file in which the prototype of these five functions are defined

counterMonitor.h

```
void CounterInit(int n);  /* initialize monitor */
int INC(void);            /* increase the counter */
int DEC(void);            /* decrease the counter */
void SET(int n);          /* reset the counter */
int GET(void);            /* retrieve counter's value */
```

Counter Monitor – counterMonitor.c

```
#include  <pthread.h>
#include  "counterMonitor.h"
/* the static counter     */
static  int  count;
/* the static mutex lock */
static  pthread_mutex_t  MonitorLock;

void  CounterInit(int  n) {
     count = n;
     pthread_mutex_init(&MonitorLock, NULL);
}

int  INC(void) {
   int  value;
   /* lock the monitor      */
   pthread_mutex_lock(&MonitorLock);
      value = (++count);
   /* release the monitor   */
   pthread_mutex_unlock(&MonitorLock);
   return  value;
}
```

```
int  DEC(void) {
   int  value;

   pthread_mutex_lock(&MonitorLock);
      value = (--count);
   pthread_mutex_unlock(&MonitorLock);
   return  value;
}

void  SET(int  n) {
   pthread_mutex_lock(&MonitorLock);
      count = n;
   pthread_mutex_unlock(&MonitorLock);
}

int  GET(void) {
   int  value;

   pthread_mutex_lock(&MonitorLock);
      value = count;
   pthread_mutex_unlock(&MonitorLock);
   return  value;
}
```

Explanations

- The shared counter, **count**, is a local variable of the monitor
  - It is declared with **static** so that it cannot be accessed outside of this file
- The lock that is used for mutual exclusion, **MonitorLock**, is also declared with **static**, since it is part of the monitor
- To make sure that there is only one thread can execute any monitor procedure, our strategy is: **once the execution enters a monitor procedure, it locks the monitor using MonitorLock, does the job, and finally releases the lock**
  - Based on this strategy, monitor procedure **INC()** locks the monitor, increases the value of the counter, and releases the lock. Finally, it returns the value saved right after the counter is increased
  - The way of writing **DEC()**, **SET()** and **GET()** is all the same
- The initialization function must be called in the main program, since a monitor implemented in this way is not able to initialize itself

Wrong Solution!

```
int  INC(void){
    pthread_mutex_lock(&MonitorLock);
    ++count;
    pthread_mutex_unlock(&MonitorLock);
    return  count;
}
```

- Suppose when INC() was called the value of count is 2
- The monitor is locked, count is changed to 3, and the monitor is unlocked
- At this moment, we would expect INC() returns 3
  - Before executing the return statement, another thread calls INC() (or DEC() or SET()) and has the value of count changed
    - This call to INC() will not return 3 but some other unexpected value
  - This is why the new counter value is immediately saved to value, which is returned after exiting the monitor

Self-initializing Monitors

```
int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

- The first call to pthread_once() by any thread in a process, with a given once_control, shall call the init_routine with no arguments
- Sub- sequent calls of pthread_once() with the same once_control shall not call the init_routine
- To make your monitor self-initializing write an initialization function with no parameters and then in every monitor function exposed to the user call pthread_once()

Readers-Writers Monitor

ReaderWriterMonitor.h

```
void start_read();
void start_write();
void stop_read();
void stop_write();
```

```
#include  <pthread.h>
#include  "ReaderWriterMonitor.h"

static pthread_once_t is_initialized = PTHREAD_ONCE_INIT;
static int readers = 0,writers = 0;
static int busy = 0;
static pthread_cond_t okRead, okWrite;
static pthread_mutex_t mLock;

extern void init () {
  pthread_mutex_init(&mLock, NULL);
  pthread_cond_init(&okRead, NULL);
  pthread_cond_init(&okWrite, NULL);
}
```

ReaderWriterMonitor.c

Readers–Writers Monitor

```
void start_read(){
   pthread_once(&is_initialized, init);
   pthread_mutex_lock(&mLock);
   while (busy)
      pthread_cond_wait(&okRead,&mLock);
   readers++;
   pthread_cond_signal(&okRead);
   pthread_mutex_unlock(&mLock);
}

void start_write() {
   pthread_once(&is_initialized, init);
   pthread_mutex_lock(&mLock);
   writers++;
   while (busy || readers > 0) {
      pthread_cond_wait(&okWrite, &mLock);
   }
   busy = 1;
   pthread_mutex_unlock(&mLock);
}
```

```
void stop_read() {
   pthread_once(&is_initialized, init);
   pthread_mutex_lock(&mLock);
   readers--;
   if (readers == 0)
      pthread_cond_signal(&okWrite);
   pthread_mutex_unlock(&mLock);
}

void stop_write() {
   pthread_once(&is_initialized, init);
   pthread_mutex_lock(&mLock);
   busy = 0;
   writers--;
   if (writers > 0)
      pthread_cond_signal(&okWrite);
   else
      pthread_cond_signal(&okRead);
   pthread_mutex_unlock(&mLock);
}
```

ReaderWriterMonitor.c

Readers-Writers Monitor

```
#include  <pthread.h>
#include  "ReaderWriterMonitor.h"

void reader(void *id) {
   while (1) {
      start_read();
      // Code that reads!
      stop_read();
   }
}

void writer(void *id) {
   while (1) {
      start_write();
      // Code that writes!
      stop_write();
   }
}
```

```
void main() {
   pthread_t r, w;
   pthread_create(&r, NULL, (void*) reader, NULL);
   pthread_create(&w, NULL, (void*) writer, NULL);
   pthread_join(r, NULL);
   pthread_join(w, NULL);
}
```

ReaderWriter.c