Chapter 1

Design of Module *Threads*

1.1 Assignment Requirement

This project your goal is to improve the thread component.

1. Improve the timer implementation by removing busy waiting.
2. Schedule threads according to their priority.
3. Solve priority inversion problems by implementing priority donation.

1.1.1 Initial Functionality

- The timer uses busy waiting: it calls *ThreadYield()* in a while loop as long as the time returned by *IomuGetSystemTime()* is smaller than the thread's trigger time.
- HAL9000 currently implements a basic round-robin scheduler which doesn't take into account thread priorities when choosing the next thread to run. It schedules threads using a simple First Come First Served technique, letting each thread run for a constant time quantum.

1.1.2 Requirements

The requirements of the "Threads" assignment are the following:

- *Timer.* You have to change the current implementation of the timer, named `EX_TIMER` and located in the file "`ex_timer.c`", such that to replace the busy-waiting technique with the sleep – wakeup (block – unblock) one. A sleeping thread is one being blocked by waiting (in a dedicated waiting queue) for some system resource to become available. You could use executive events (`EX_EVENT`) to achieve this.

- *Priority Scheduler — Fixed-Priority Scheduler.* You have to change the current Round-Robin (RR) scheduler, such that to take into account different priorities of different threads. The scheduling principle of the priority scheduler is "any time it has to choose a thread from a thread list, the one with the highest priority must be chosen". It must also be preemptive, which means that at any moment the processor will be allocated to the ready thread with the highest priority. The priority scheduler does not change in any way the threads' priorities (established by threads themselves) and for this reason could also be called fixed-priority-based scheduler.
- *Priority Donation.* As described in the documentation after the fixed priority scheduler is implemented a priority inversion problem occurs. Namely, threads with medium priority may get CPU time before high priority threads which are waiting for resources owned by low priority threads. You must implement a priority donation mechanism so the low priority thread receives the priority of the highest priority thread waiting on the resources it owns.

1.1.3 Requirements allocation

TEAM HAPPY CODING (3-members teams)

1. Timer - Palko Lorand Arpad
2. Priority Scheduler: Szabo Cristian-Iacob
3. Priority Donation: Rusu Vlad

1.1.4 Basic Use Cases

The Executive Timer is used for thread scheduling. Each thread runs for a certain time, and the block-unblock mechanism makes use of this timer.

The Priority Scheduler is used for scheduling threads on the CPUs. Priority donation is needed due to priority inversion.

1.2 Design Description

1.2.1 Needed Data Structures and Functions

ex_timer.h
struct _EX_TIMER {
...
EX_EVENT Event;
}

timer.c
struct _EX_TIMER_SYSTEM_DATA {
LIST_ENTRY AllTimers;
}
    void notifyTimer()
- checks if any of the global timers fired

iomu.c
void _IomuSystemTickInterrupt()
- call *notifyTimer()*

thread_internal.h
struct _THREAD {
...
THREAD_PROIRITY StaticPriority;

LOCK HeldMutexesLock;

__Guarded_by(HeldMutexesLock)
LIST HeldMutexes;
...
}

thread.c
struct _THREAD_SYSTEM_DATA {
...
LIST_ENTRY AllReadyThreads[32];
...
}

THREAD_PRIORITY ThreadGetPriorityRecursive(PTHREAD Thread, BYTE RecursionDepth);

1.2.2 Detailed Functionality

1. Timer
    - The main idea is to hold a list of initialized timers that will be checked at each time tick. The threads for which the time expired will be signaled using their associated *EX_EVENT*. The list of timers must be synchronized using a lock in order to avoid race conditions when multiple threads try to create a timer at the same time.
    - The timer mechanism makes use of the *_IomuSystemTickInterrupt* ISR to check if any of the timers fired.

- A call *ExTimerWait()*, basically forwards the call to *ExEventWait()* which will put the current thread to sleep.
- The event associated to the timer will be a notification type event, because there may be more than one thread waiting for the same timer.
- The *EX_TIMER* structure will be augmented to hold an *EX_EVENT* object
- *ExTimerInit()* will create a new timer object and a new notification type *EX_EVENT* associated with it. Then the new timer will be added into the global list of timers.

2. Priority scheduler

   - The *THREAD_SYSTEM_DATA* structure will be augmented to keep a list of waiting lists, each index corresponding to a priority level: list[0] = list of threads with priority 0 ... list[31] = list of threads with priority 31. Threads will be added to the lists corresponding to their priority whenever *ThreadInit()* is called.
   - The scheduler will traverse the list from index 31 (highest priority) to 0 (lowest priority) and schedule the threads from these waiting lists, so the highest priority thread will be always scheduled first.
   - In order to implement priority donation, we need to keep for each thread a list containing all the mutexes the thread holds. This list will be added to the *_THREAD* structure together with a static priority field that will store the thread's initial priority. We also need to keep a reference to each mutex's owner and a list of waiting threads for each mutex, but these are already present in the *_MUTEX* structure. This way, if a thread is waiting for a mutex, we can check the mutex's owner to know who will receive the donation and we will have access to all threads waiting for that mutex in order to compute the donated priority.
   - A thread's priority is computed as the maximum value of its static priority and its donated priority. The donated priority's value will be determined using a *ThreadGetPriorityRecursive()* function which will compute the maximum value from all held mutex's waiting lists recursively. In order to make sure that the function returns, a max recursion depth will be set (8). The *_THREAD* structures will be updated whenever *MutexAcquire()*, *MutexTryAcquire()* and *MutexRelease()* are called.
   - When *MutexAcquire()* is called, the owner thread of the mutex is set and the mutex is inserted into the thread-held mutexes list. The nested donations problem is solved by *ThreadGetPriorityRecursive()* which queries the thread's priorities in depth, having above recursion depth of 8.

- When *MutexRelease()* is called, the owner thread of the mutex is set to NULL and the mutex is removed from the corresponding thread-held mutexes list. The mutex's waiting list will be than traversed and the thread with the highest priority will be unblocked if it has a higher priority than the current thread. Unblocking the thread will be done through a call to *ThreadYield()* and the thread will be scheduled.
- *ThreadSetPriority()* won't lead to any race conditions as we calculate the thread's priority as the maximum value of its static priority and its donated priority which is computed on the fly through the *ThreadGetPriorityRecursive()* function.

1.2.3 Explanation of Your Design Decisions

- The timer implementation with *EX_EVENT* was chosen because Executive Events are already implemented in HAL9000. All we have to do is to forward *ExTimerWait()* to *ExEventWait()*, register an event for each timer and keep a global list with all the initialized timers. We fire each timer when its time slice expires.
- For priority scheduling, keeping a list of waiting lists for each priority level seems to be the simplest solution that should not generate any unexpected problems. Computing the donated priority using the *ThreadGetPriorityRecursive()* should solve the priority donation problem with ease.

1.3 Tests

1.4 Observations