Lab 18 Real Time Preemptive Multi-Treaded Operating System

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals
- Design an interrupting interface,
- Use the 6812 real time interrupt feature,
- Design a round robin multi-thread scheduler,
- Implement blocking/counting semaphores,
- Implement inter-thread communication.

Review
- Valvano Chapter 5 on Threads,
- Sections 4.1, 4.2, 4.5, 6.4, and 6.5.1 of Operating System Concepts by Silberschatz and Galvin, 1994.

Starter files
- THREAD3.C, SCI12.H, SCI12A.C

Background

Your job is to evaluate then expand operating system commands that implement a multiple thread environment. A thread is a "light weight" process. Threads share resources (global memory, I/O devices) but have separate registers, stack and local variables. Typically threads cooperate to achieve a common goal.

It will be important to implement reentrant code, because multiple threads will be executing the same software. You will have to carefully consider what information is local to each thread and what is global. A nonreentrant subroutine will have a section of code called a **vulnerable window**. An error occurs if

1) one thread calls the nonreentrant subroutine
2) is executing in the "vulnerable" window when interrupted by a second thread
3) the second thread calls the same subroutine
4) control is returned to the first thread
5) the first thread finishes the subroutine.

A vulnerable window may exist when two different subroutines access the same memory-resident data structure. Consider the situation where two concurrent threads are communicating uses a FIFO queue. What would happen if the PUT subroutine executes in between any two instructions of the GET routine (or vice versa?)

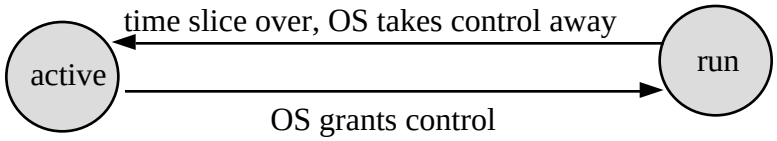
Description of First Part

For the first part of this lab you will evaluate and make minor modifications an existing multithreaded system so that you can "visualize" the execution pattern of the system. The initial system has exactly 2 producer threads and 1 consumer thread that are created at compile/assembly time. It uses spinlock semaphores to provide thread synchronization. These threads will run indefinitely. You will not need a dynamic memory manager, because the thread control block, **tcb**, for each thread can be allocated statically. The **tcb** contains information for each thread:

- Thread number (not really needed, but is used for debugging)
- Current Stack Pointer, SP, for this thread
- Stack area for this thread

Inside the stack area, the local variables are stored. When a thread is suspended because of a time slice interrupt, the registers (CCR,A,B,X,Y,PC) are stored also on this stack. If a thread wishes to output to the display, but another thread is currently outputting, it will spin (do nothing). In the first part, we will use a spinlock semaphore mechanism to implement the synchronization among multiple threads.

If a thread needs information from a FIFO (calls GET), then it will spin if the FIFO is empty (because it can not retrieve any information.) On the other hand, if a thread outputs information to a FIFO (calls PUT), then it will spin if the FIFO is full (because it can not save its information.)



A thread can be in one of two states. A thread is in the **active state** if it ready to run but waiting for its turn. A thread is in the **run state** if it currently executing. With a single instruction stream computer like the 6812, at most one thread can be in the run state at a time. A circular linked list data structure holds the ready and active threads.

The 6812 output compare interrupt feature (OC3) will be used by your operating system (OS) to grant and take away execution control from the available active foreground threads. You will implement pre-emptive round robin scheduling. Additional output computer handlers will be added in the second part of this lab to measure elapsed time and execute time-critical periodic tasks. These other output compare interrupts do not participate in the scheduling of foreground threads.

When passing data between two foreground threads, we can use a buffered approach (FIFO) or an unbuffered (mailbox) approach. When using a FIFO buffer, the two counting semaphores called, `Available` and `RoomLeft`, contain the number of 8-bit entries currently stored in the message FIFO and the number of 8-bit spaces left in the FIFO respectively. `Available` is initialized to zero, and `RoomLeft` is initialized to the maximum allowable number of elements in the FIFO. The **Put** routine executes the following steps:

```
Wait(&RoomLeft)
Disable Interrupts
Enter 8-bit data into the FIFO structure
Enable Interrupts
Signal(&Available)
```

The **Get** routine executes the following steps:

```
Wait(&Available)
Disable Interrupts
Remove 8-bit data from the FIFO structure
Enable Interrupts
Signal(&RoomLeft)
```

The "producer" thread creates data (using its local variable i++), then sends the data to a consumer (calls Put). The "consumer" thread receives the data from a producer (calls Get). Occasionally the "producer" and "consumer" threads will call Message(). The rate of these periodic messages was chosen high enough to provide some visual feedback that the system is running, but low enough to prevent the message handling from being a major bottleneck itself to the system bandwidth. The producer and consumer functions should be reentrant. In this way, multiple threads can be concurrently running the same program. The local variables exist on the stack. The values of the local variables need to be saved and restored when a thread is suspended and resumed. In this way, the local variables are not accessible by other threads. The FIFO is used for interthread communication. The complicating situation is that there are multiple threads active at one time communicating with the single FIFO.

When one of the threads involved in the buffered producer/consumer communication is a background thread, we must remove the `Wait` call from the ISR. This is because only foreground threads will be allowed to spin or block. The buffered input has the following semaphore usage. The **RxPutFifo** routine is called from the SCI ISR when new data is received. In this example, data is lost if the RxFifo becomes full.

```
Input the new input data from the SCI receiver
Try to enter 8-bit data into the RxFifo
If successful Signal(&RxAvailable)
```

The **RxGetFifo** routine is called when a foreground thread calls `InChar`

```
Wait(&RxAvailable)
Remove 8-bit data from the RxFifo
```

Similarly, the buffered output has the following semaphore usage. The **TxGetFifo** routine is called from the SCI ISR when the output channel is idle (ready for more output). In this example, the output interrupts are disarmed if the TxFifo becomes empty.

```
Try to remove 8-bit data from the TxFifo
If successful Signal(&TxRoomLeft) and output the data to the SCI transmitter
If not successful, disarm
```

The **TxPutFifo** routine is called when a foreground thread calls `OutChar`

```
Wait(&TxRoomLeft)
Enter the 8-bit data into the TxFifo
```

In this next example, we will pass data from a single foreground producer thread to a single foreground consumer thread using an unbuffered approach. A single global (called mailbox) will contain the data passed from producer to consumer. The counting semaphores, `Avail Ack`, are both initialized to zero meaning the mailbox is empty. If the producer executes **Send** first, it will put its data into the mailbox, increment `Avail` then wait on the semaphore `Ack`. When the consumer arrives second (executing **Receive**), it will decrement `Avail`, get the data and increment `Ack`. Incrementing `Ack` will allow the producer to proceed. If the consumer executes **Receive** first, it will wait on the semaphore `Avail`. When the producer arrives second (executing **Send**), it will put its data into the mailbox, increment `Avail` then wait on the semaphore `Ack`. Incrementing `Avail` will allow the consumer to proceed.

When the consumer proceeds, it will get the data and increment `Ack`. Incrementing `Ack` will allow the producer to proceed. The **Send** routine executes the following steps:

```
Put data into the mailbox
Signal(&Avail)
Wait(&Ack)
```

The **Receive** routine executes the following steps:

```
Wait(&Avail)
Retrieve the data from the mailbox
Signal(&Ack)
```

When we perform SCI serial port I/O we will need a mechanism to share this resource. You will have to implement mutual exclusion (only one thread at a time can call the SCI12 functions.) A traditional computer science term for this type of semaphore is mutex. We will call our semaphore DisplayFree. It will prevent more than one thread from outputting at the same time. It is initialized to 1 that means there is 1 display available. When the DisplayFree semaphore is zero, it means no displays are free (a thread is currently doing output.) Some operating systems provide special support for this true/false type of semaphore, calling it a binary semaphore. For example if you wished to output a message, then a thread could call a function like the following:

```
void Message( char Name, unsigned int data){
        Wait(&DisplayFree);
        OutChar(Name);
        OutUDec(data);
        OutChar(CR);
        Signal(&DisplayFree);
}
```

First Part Preparation (do this before your lab period)

1) Run the THREAD3.C program (get the latest version from the web page). Observe the waveform patterns on Port J and Port H using the logic analyzer or scope. Draw a sketch of PJ2, PJ1, PJ0, PH5, PH4, PH3, PH2, PH1, PH0 during execution. Include enough data so that the execution pattern is clear. Label the plot, describing the execution pattern. Draw something similar to Figure 5.7 in the book. In particular, what does it mean when Port H bit 4 is high?

2) Modify the THREAD3.C program so that the system bandwidth can be measured quantitatively. We define bandwidth in this example as the average rate (in bytes/sec) at which data is entered into the FIFO. For this example in the steady state, the average rate of data entering the FIFO equals the average rate of data leaving the FIFO. Why is it true in this case? Give a situation in which the average rate at which `Put()` is called is different from the average rate `Get()` is called.

3) Run the modified THREAD3.C program for about 5 different FIFO sizes. Vary the FIFO size (`FifoSize`) from 2 to 50 elements and plot the system bandwidth as a function of FIFO size. Explain your results in two or three sentences.

4) Run the modified THREAD3.C program for about 5 different timeslice's. You should vary the timeslice from about 500 to 50000 cycles and plot the system bandwidth as a function of timeslice. Explain your results in two or three sentences.

5) Look at the assembly listing generated by the compiler. Use this information to draw a picture that includes the 3 values of `Id` located in the tcb's of each thread and the 3 values of the local variable `i` located on the stacks of each thread. Also draw the 6812 hardware stack pointer, S, and index register, X, for the thread that is running. Draw something similar to Figure 5.6 in the book.

First Part Checkout (show this to the TA)

1) Show your visualization method to the TA,
2) Demonstrate your method to measure bandwidth to the TA,
3) Explain to the TA the assembly listing of the thread switcher, and stack pictures,
4) Be prepared to discuss the data you collected in the first part.

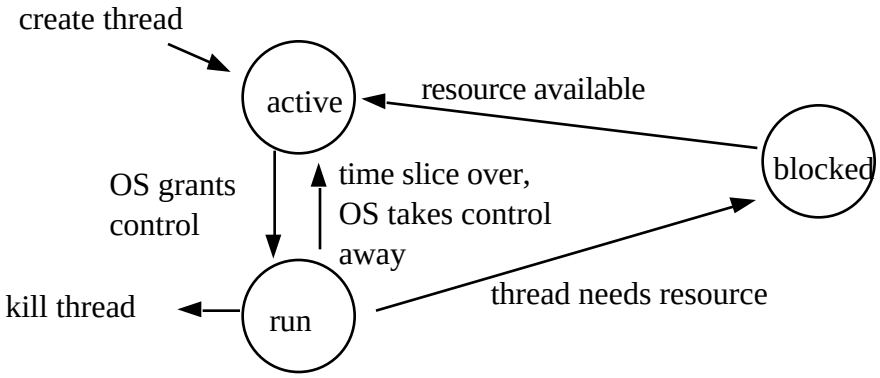
Description of the Second Part

The second part must be completed in addition to the first part. The basic idea of the second part is to replace the spinlock semaphores with blocking semaphores. Now, a thread can be in one of three states. A thread is in the **blocked state** when it is waiting for some external event like input/output (keyboard input available, printer ready, I/O device available.) If a thread communicates with other threads then it can be blocked waiting for an input message or waiting for the another thread to be ready to accept its output message. Both types of blocking that will be implemented in the part of this lab.

If a thread wishes to output to the display (e.g., the thread calls Message), but another thread is currently outputting, it will block. We will use a blocking semaphore to implement the sharing of the display output among multiple threads.

If a thread needs information from a FIFO (calls GET), then it will be blocked if the FIFO is empty (because it can not retrieve any information.) On the other hand, if a thread outputs information to a FIFO (calls PUT), then it will be blocked if the FIFO is full (because it can not save its information.)

A good implementation is to use linked list data structures to hold the ready and blocked threads. You will wish to create three blocked linked lists, one for blocked waiting for the output display to be free, one for FIFO full during a PUT and one for FIFO empty during a GET. In general, we will have one blocked list with each blocking semaphore.



The THREAD3.C program you analyzed in the first part used a simple signed integer for the counting spinlock semaphore. In this second part, you will extend the semaphore to a structure that contains both the semaphore value and a pointer to a tcb list containing threads that are blocked on the semaphore. For example, we could define the semaphore type as

```
#define null 0
struct sema4
{   int value;
    struct TCB *BlockPt;     /* pointer to threads blocked on this  */
};
typedef struct sema4 sema4Type;
typedef sema4Type * sema4Ptr;
```

To create a semaphore, we would:

```
semaphoreType DisplayFree; /* Semaphore used to share Display */
```

Io initialize the semaphore, you could call a function like the following:

```
void Initialize(sema4Ptr semaphore, int InitialValue ){
      semaphore->value=InitialValue;
      semaphore->BlockPt=null;
```

Message, Producer, Consumer, Put and Get would remain exactly the same. In `InitFifo`, we would use the new syntax (like the above main example) to initialize the semaphores. In wait and signal, we pass in a pointer to the semaphore (instead of a pointer to an integer). If you were still doing spinlock (but you are not) then the new syntax would be:

```
void Signal(sema4Ptr semaphore){
     asm(" sei");  /* increment is atomic */
     PORTH|=0x20;
     (semaphore->value)++;
     PORTH&=0xDF;
     asm(" cli");}
```

Just like the first part, the main program is only used to initialize the OS and launch the first thread. Your operating system (the preemptive thread scheduler) will control when and which threads will be allowed to execute. You will maintain at least four linked-lists TCB queues: one queue for the active threads and three queues for the blocked threads. You will use a real time interrupt (e.g., OC3) to switch control between the active threads. You will implement equal priority (round robin) with bounded waiting among the threads. This means the OS will cycle through the active threads and execute each one in turn. Similarly, if multiple threads are blocked, when it is time to wakeup a thread, the OS will wakeup the one that has been blocked the longest. If a thread requests a resource that is unavailable, i.e.,

tries to perform display output, while another thread is currently outputting
calls PUT when the FIFO is full
calls GET when the FIFO is empty

your system should move the thread to the appropriate blocked linked-list. Careful thought should go into when to remove a thread from the blocked list. Just like the first part, the threads never die. They are created at compile time, and execute in a continuous loop. Careful thought should go into what information should be placed into the **tcb** of each thread (e.g., register values, program counter, local variable space etc.) The space for the tcb's is allocated statically and never released.

Your operating system will implement at least three counting/blocking semaphores. A semaphore has an integer global variable (an up/down counter), a blocked **tcb** linked list and three functions: Initialize, Wait, and Signal. The following steps should be taken as a starting point for the design of your semaphore system. You must be careful to block and wakeup the correct number of threads.

Second Part Preparation (do this before your lab period)

Show the syntax-free software that implements the blocking semaphores.

Second Part Procedure (collect this data)

1) Run the blocking semaphore system for the same 5 FIFO sizes as you performed in the first part. Again plot the system bandwidth as a function of FIFO size. Compare your results of the blocking semaphore system to the spinlock semaphore system.

2) Run the blocking semaphore system for the same 5 timeslice's. Again plot the system bandwidth as a function of timeslice. Compare your results of the blocking semaphore system to the spinlock semaphore system.

Second Part Checkout (show this to the TA)

Show the operation of the blocking semaphore system. Demonstrate your method to visualize the real time execution pattern. For one interrupt rate (timeslice), demonstrate your system with 2 different FIFO sizes. Discuss your results obtained in the second part procedure.

Hints

1) Make small changes and save the changes using new file names, so that when something doesn't work you can go back to a version that does work and try something new.
2) You will have to debug this system in small parts. A mechanism to visualize the real time execution will be helpful.
3) Avoid infinite loops with the interrupts disabled (a crash).
4) Avoid using breakpoints and single stepping. Remember to use the nonintrusive debugging techniques that you have developed. If you store data into global memory, the information should be available for viewing even after a crash or a hardware reset. (Interesting note: because the TExaS simulator models both the hardware and software, breakpoints and single stepping are appropriate in this setting. A simulator breakpoint halts both the external hardware and software.)
5) Observe the assembly code generated by the compiler for the OC3 routine, and for the Wait function. It will be important to create the stack to "look" like an OC3 interrupt when suspending the thread being blocked. For example, the compiler will allocate space for parameters and local variables on the stack. The compiler may allocate its own local variables when implementing complex expressions, or calling other functions with stack parameters. Remember that a blocked thread will be suspended by the Wait function, but will be launched again (after wakeup)

by the OC3 handler. You may wish to implement all or part of Wait by calling swi, and writing a SWI handler. The software interrupt will push registers like an OC3 interrupt, but you still must be very careful.

6) The compiler may allocate local variables within the OC3 handler (even if you didn't explicitly define any yourself.) This causes the data to be allocated on one stack and deallocated on another.

7) Run the thread examples on the TExaS simulator. There are implementations in 6812 assembly, ICC12 C, Hiware 6812 C, 6811 assembly and ICC11 C. To transfer simulator implementation to the actual physical system, only the SCI baud rate needs to be changed.

8) Pathway to success

Run the thread example included
Get a good idea of where the stacks are, where the local variables are
Change the semaphore from integers to a structure (but don't use the pointer yet)
Implement Blocking semaphore for printing (test the blocking part separately)
Convert the other semaphores to blocking

Look for the most recent files on the network.