Lab 17 Real Time 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
- Develop OS facilities for real time applications,
- Coordinate multiple foreground and background threads,
- Design a round robin multi-thread scheduler,
- Blocking semaphores and priority scheduler,
- Implement inter-thread communication.

Review
- Valvano Chapter 4 on Fifo queues and interrupts,
- Valvano Chapter 5 on Threads and semaphores,
- Valvano Chapter 6 on periodic interrupts using output compare

Starter files
- Many files with names Lab17*.*

Background

Your job is to evaluate then expand operating system commands that implement a multiple thread environment. In real time applications, the scheduling of software tasks is critical for the proper operation of the system. For a data acquisition system, the software must start the analog-to-digital converter (ADC) and read the result at precise time intervals. Let $\Delta\mathbf{t}$ be the time interval, which is one divided by the sampling rate.

$$\Delta\mathbf{t}=1/\mathbf{f}_s$$

For a control system, the software must read the sensors, perform the digital control equations, then output to the actuators at a fixed rate. For a system that generates signals, the software must output to the digital-to-analog converter (DAC) at a fixed rate. We can define time-jitter, $\delta\mathbf{t}$, as the difference between when a periodic task is supposed to be run, and when it is actually run. Let $\mathbf{t_n}$ be the time the software task is actually run, and let $\mathbf{n}\Delta\mathbf{t}$ be the time it was supposed to be run, then the time-jitter at sample **n** is

$$\delta\mathbf{t_n}= \mathbf{t_n} - \mathbf{n}\Delta\mathbf{t}$$

For a real time system with periodic tasks, we must be able to place an upper bound, **k**, on the time-jitter.

$$-\mathbf{k} \le \delta\mathbf{t_n} \le +\mathbf{k} \quad \text{for all } \mathbf{n}$$

Sometimes it is more important to control the time difference between periodic events rather than the absolute time itself. Let $\Delta\mathbf{t_n}$ be the actual time difference between two executions of a software task (e.g., starting the ADC). The desired time difference is $1/\mathbf{f}_s$. For this situation, we define the time-jitter at sample **n** to be

$$\delta\mathbf{t_n}= \Delta\mathbf{t_n} - 1/\mathbf{f}_s$$

Again, we must be able to place an upper bound, **k**, on the time-jitter.

$$-\mathbf{k} \le \delta\mathbf{t_n} \le +\mathbf{k} \quad \text{for all } \mathbf{n}$$

For example, in this lab the ADC is sampled at 2000 Hz. This means the ADC should be activated every 125µs. This sampling rate is fixed and should not be increased or decreased. Using a **ScanPoint**, the following TheLog.RTF data shows the TCNT values measured at the first ten times the ADC was sampled.

```
_Time=1 TCNT=16660
_Time=2 TCNT=20651
_Time=3 TCNT=24653
_Time=4 TCNT=28652
_Time=5 TCNT=32655
_Time=6 TCNT=36673
_Time=7 TCNT=40652
_Time=8 TCNT=44723
_Time=9 TCNT=48652
_Time=10 TCNT=52651
```

Although the difference should have been exactly 4000 cycles (500 µs), there was some variability in when the signal was sampled. In this system, the measured time-jitter is usually less than ±20µs. Under most situations, this error is acceptable, and we are confident to specify this system as real time.

For a real time system with input/output devices, the software latency is important. For an input device, the software latency is the time delay between when the hardware says the input is ready and the time when the software reads the data. With a simple serial port like ones on the 6812, the software must response to an input within 10 bit times or risk an overrun error (lost data.) For an output device, the software latency is the time delay between when the hardware says the output is idle and the time when the software write new data to the device. The software

latency for the output task will affect the overall bandwidth, but there is usually no hard upper bound above which the system stops working.

Not all software tasks in a real time system require execution at specified times. For example, in a data acquisition and control systems, updating visual displays or saving the results in secondary storage can often be performed when the computer is free, i.e., not needed for time critical functions.

When there are a small number of real time tasks in a system, a simple software solution can usually be found. But as the number and complexity of the tasks increase, we will need a set of OS facilities to manage time. Foreground threads will be run using a priority thread scheduler. Background threads (interrupt service routines) will be run as a result of specific hardware conditions. In particular, SCI transmit, SCI receiver, and output compare will generate hardware interrupts in this lab. 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. Background threads will have the highest priority and they will be used to perform the most time-critical functions. It will be important to develop debugging tools to visualize software activity. Using these tools, you will develop performance measures to evaluate system efficiency. You will first add cooperative multitasking, then you will extend the thread scheduler to implement blocking. In this way important foreground tasks can be run only when there is work to be done. Priority can be added so that more important software tasks are performed first. Semaphores will be used with the OS to provide synchronization/communication between threads.

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

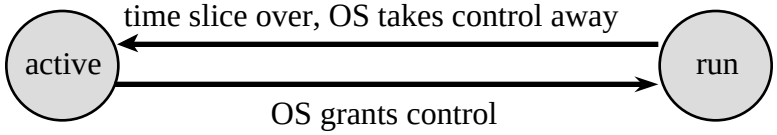
1) one thread calls the nonreentrant subroutine
2) is executing in the "vulnerable" window when interrupted by a second thread
3) the second thread calls the same subroutine
4) control is returned to the first thread
5) the first thread finishes the subroutine.

A vulnerable window may exist when two different subroutines access the same memory-resident data structure. Consider the following situation: `Thread1` increments a counter, and `Thread2` decrements it. Because the ICC12 compiler implements the increment and decrement using multiple instructions, the read-modify-write access is nonatomic, and hence a vulnerable window exists. In particular, if `Thread1` is interrupted by `Thread2` in the middle of a read-modify-write access to the counter, then the counter will have an incorrect value.

```
int counter; // shared global
```

```
void                    | void
Thread1(void){          | Thread2(void){
  while(1){             |   while(1){
    counter++;          |     counter--;
  }                     |   }
}                       | }
```

Look at the Lab17.C example. When `SCI_InChar()` needs information from its RxFifo, it calls `RxFifo_Get`. If the RxFifo is empty, it will spin on the semaphore `RxAvailable` because it can not retrieve any information. On the other hand, when `SCI_OutChar()` outputs information to its TxFifo, it calls `TxFifo_Put`. If the TxFifo is full, then it will spin on the `TxRoomLeft` semaphore because it can not save its information. The spinning occurs in the `OS_Wait` routine. In the third part of this lab, you will replace this inefficient do-nothing software function with a blocking scheme, which will release the computer to execute other real tasks. But for now, all threads remain in the ready queue.



With spinlock semaphores, a foreground thread can be in one of two states. A foreground thread is in the **active state** if it ready to run but waiting for its turn. A foreground thread is in the **run state** if it is currently executing. With a single instruction stream computer like the 6812, at most one thread can be in the run state at a time. Therefore, the thread that is running uses the actual registers (CCR, A, B, X, Y, SP, and PC.) On the other hand, a foreground thread that is not running has its registers (CCR, A, B, X, Y, and PC) on top of its stack, and has its stack pointer saved in its TCB. A circular linked list data structure holds the ready and active threads. Again, the background threads are the interrupt service routines, which are executed in response to specific hardware events.

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. A second output compare channel (OC4) is used by the Producer to implement the real-time data acquisition.

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

```
OS_Wait(&DataRoomLeft)
Disable Interrupts
Enter 8-bit data into the DataFifo structure
Enable Interrupts
OS_Signal(&DataAvailable )
```

The **Receive** routine, called by the `Consumer`, executes the following steps:

```
OS_Wait(&DataAvailable)
Disable Interrupts
Remove 8-bit data from the DataFifo structure
Enable Interrupts
OS_Signal(&DataRoomLeft)
```

A "producer" thread creates data, then sends the data to a consumer (calls **Send**). A "consumer" thread receives the data from a producer (calls **Receive**). The `DataFifo` is used for interthread communication. In some applications there might be multiple producers and multiple consumers. In this lab however, we will have the simple situation of having a single producer and a single consumer.

When one of the threads involved in the buffered producer/consumer communication is a background thread, we must remove the `OS_Wait` call from the ISR. This is because only foreground threads will be allowed to spin or block. In the Lab17.C example, the buffered input has the following semaphore usage. The **RxFifo_Put** 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 OS_Signal(&RxAvailable)
```

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

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

Similarly, the buffered output has the following semaphore usage. The **TxFifo_Get** 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 OS_Signal(&TxRoomLeft) and output the data to the SCI
If not successful because it was empty, disarm
```

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

```
OS_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. You will not be using this unbuffered approach. It is included here so that you can compare it with the buffered approach. A single global (called `mailbox`) will contain the data passed from producer to consumer. The counting semaphores, `Available Acknowledge`, 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 `Available` then wait on the semaphore `Acknowledge`. When the consumer arrives second (executing `Receive`), it will decrement `Available`, get the data from the `mailbox` and increment `Acknowledge`. Incrementing `Acknowledge` will allow the producer to proceed. If the consumer executes `Receive` first, it will

wait on the semaphore `Available`. When the producer arrives second (executing `Send`), it will put its data into the `mailbox`, increment `Available` then wait on the semaphore `Acknowledge`. Incrementing `Available` will allow the consumer to proceed. When the consumer proceeds, it will get the data and increment `Acknowledge`. Incrementing `Acknowledge` will allow the producer to proceed. The `Send` routine executes the following steps:

```
Put data into the mailbox
OS_Signal(&Available)
OS_Wait(&Acknowledge)
```

The `Receive` routine executes the following steps:

```
OS_Wait(&Available)
Retrieve the data from the mailbox
OS_Signal(&Acknowledge)
```

An alternative solution to the unbuffered `mailbox` is essentially a fifo of size 1. In this case, `RoomLeft` and `Available` are initialized to 1 and 0 respectively. The alternative approach will have a higher bandwidth than the first mailbox implementation, because the producer waits before writing into the `mailbox` rather than after. The `Send` routine executes the following steps:

```
OS_Wait(&RoomLeft)
Put data into the mailbox
OS_Signal(&Available)
```

The `Receive` routine executes the following steps:

```
OS_Wait(&Available)
Retrieve the data from the mailbox
OS_Signal(&RoomLeft)
```

Again, you will not be using any unbuffered approaches. The description is included here for your information only.

When we perform SCI serial port output 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 SCI output 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 letter, unsigned int data){
        SCI_Open();  // calls OS_Wait(&DisplayFree); in OS
        SCI_OutChar(letter);
        SCI_OutUDec(data);
        SCI_OutChar(CR);
        SCI_Close();  // calls OS_Signal(&DisplayFree);
}
```

Before you begin writing code for this lab, you will run an existing multithreaded system and "visualize" the execution pattern of the system. The initial system has three foreground threads and two background threads. One background thread performs data acquisition (`Producer`). The other background thread uses SCI interrupts to perform serial I/O. The foreground threads are a display thread (`Consumer`), a thread calculating square roots (`Math`), and an interpreter thread (`Interpreter`). These threads are defined at compile/assembly time, and you will not need to create threads dynamically at run time. For this lab, 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).

The initial system uses spinlock semaphores to provide thread synchronization. These threads will run for a finite amount of time, then display performance measurements. There are three critical measurements in this real-time system. The first performance measure is time-jitter. As presented earlier, this initial system has a time jitter of about ±20µs on its ADC sampling. The second measure is the number of lost data points. If the consumer is waiting for the SCI port, the `DataFifo` might fill up, and the `Producer` would have no place to put the results. Lost data is particularly a problem when the `Interpreter` is being used. The last performance measurement is the number of mathematical calculations completed. This is the least important, but does give us a good indication of the efficiency of the operating system.

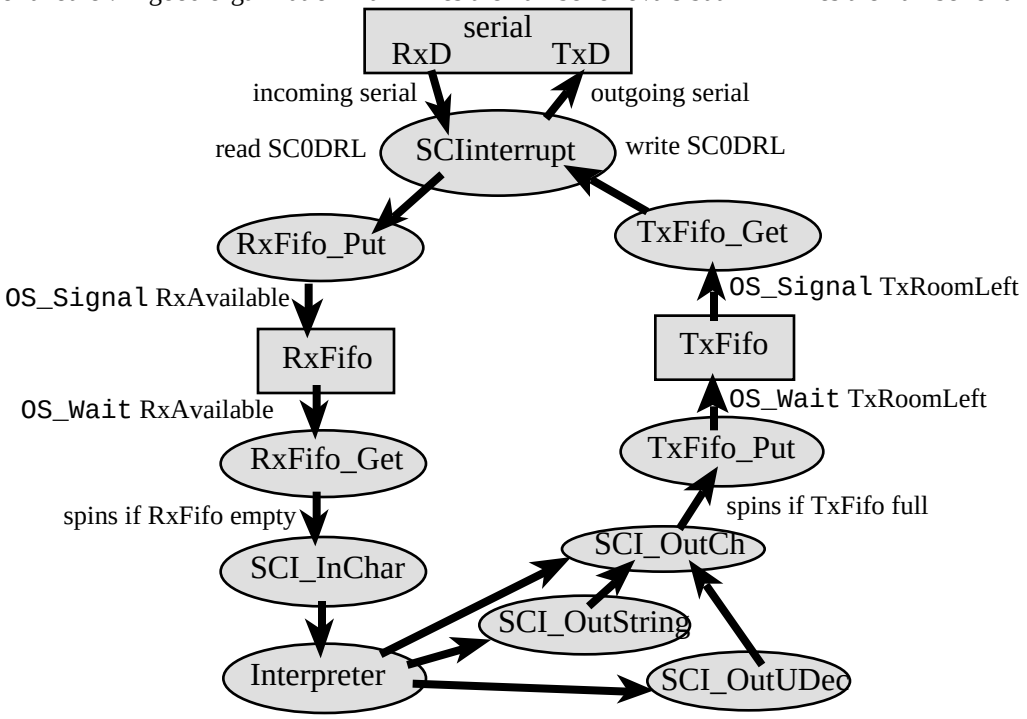
Preparation (do this before your lab period, no software development is required)

1) Install the licensed version of TExaS that came with the book. Download the latest upgrade for the TExaS simulator from the web page, and install the upgrade into the same folder as the original licensed version. Get the latest version of the Lab17*.* programs from the web page. The system includes C files, H files, and TExaS simulator files. This lab may be developed on a real 6812 or on the TExaS simulator. The starter files are configured for the simulator. To run the Lab17 programs on a real 6812 change the following definitions to

```
#define LENGTH 130        // find min/max for this many samples
#define BR 13             // Baud rate 38,400 bps
#define RUNLENGTH 20000   // display results and quit when Time==RUNLENGTH
```

If you are simulating, you first start the TExaS application, then open the microcomputer file Lab17.uc. Next, you execute `OpenS19`… and import the Lab17.S19 object code. Run the initial system and observe the waveform patterns on Port J and Port H using a logic analyzer or scope. Capture and print the logic analyzer trace of PJ2, PJ1, PJ0, PH4, PH3, PH2, PH1, PH0 during the first 4 ms of execution using PixWizard. Include enough data so that the execution pattern is clear. Read through the source code to label the significance of each of the eight debugging signals. I.e., what does it mean when each of the eight signals is high?

2) Draw a call-graph of the system. There are three hardware modules in this system. The timer (OC3, OC4, TCNT etc.), the serial port, and the ADC. Ignore the parallel ports used for debugging. Draw the hardware modules as rectangles. Represent each of the five software modules as an oval. There is a software module for each C file (Lab17, OS, SCI, RxFifo, TxFifo). Partition the ovals into public and private areas. Place names of the public/private variables/functions in the appropriate areas. Draw arrows showing where one module calls the public functions of another. A good organization maximizes the number of ovals but minimizes the number of arrows.



3) Next, draw a data-flow graph for the producer/consumer data channel. Similar to the call-graph, hardware modules and globals are rectangles. In particular, draw the `DataFifo` memory buffer as a rectangle. Different from the call-graph, we will draw a separate oval for each function. Only include the software functions that handle the data. Arrows represent the direction of data flow, not which function called which. Other synchronizing functions

like `OS_Signal` and `OS_Wait` can be added on the side as comments. The above figure shows a data-flow graph as the interpreter inputs a single character, processes that characters and sends results back through the serial port. The data does not flow through the `OS_Signal OS_Wait SCI_Open` and `SCI_Close` routines. Therefore, these functions are left off the data-flow graph.

4) Next, look at the assembly listing generated by the compiler. You should also run the software until each thread has executed at least once, then use the debugger to dump the contents of the three TCB's. 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 `me` 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.

5) There are categories into which tasks may fall. First, there are **I/O bound** tasks, where the bandwidth (data processed each second) is limited by the I/O device. For example, in a data-entry task, it usually doesn't matter how fast the computer is, the amount of information entered into the system is limited by the input typing rate of the operator. In a similar fashion, the number of pages printed per second is usually limited by the printer speed, and not by the speed of the computer. The second category describes tasks with **fixed bandwidth**, and not limited by either software or hardware. For example, the weatherman collects temperature data every hour. Temperature measurements once an hour are all we need, so a faster ADC converter, a faster temperature sensor, or a faster computer would not enhance the performance of this system. The third category, **CPU bound**, describes tasks that are limited by the execution speed of the software. For these systems, a better software algorithm, a better compiler, and a faster computer will enhance the performance. There are three tasks in this system 1) data acquisition using `Producer` and `Consumer`, 2) operator interaction using `Interpreter`, and 3) calculations using `Math`. Categorize the type of these three tasks.

6) There is a `#define DEBUG 1` definition at the top of Lab17.C. Change it to `#define DEBUG 0`, recompile and run the system again. This new system has all the debugging instruments removed. What conclusions can you make about the intrusiveness of these instruments? Re-enable the debugging instruments for the rest of the lab. I.e., change it back to `#define DEBUG 1`.

First Part Procedure (no preparation is required)

Data is lost because the Consumer fills the TxFifo and spins waiting because the TxFifo is full. Increase the size of the TxFifo and DataFifo until no data is lost. Test the system with operator input to the interpreter too. Make a table showing the three performance parameters (time-jitter, number of data points lost, number of math calculations performed) versus the size of the two fifos. Take these measurements when no input occurs in the interpreter.

The logic analyzer trace you captured in the preparation should have illustrated that this system spends a lot of time spinning in the `OS_Wait` routine. This behavior is common in I/O bound systems. Cooperative multitasking requires the tasks to suspend themselves, allowing other tasks to also run. With a preemptive scheduler, threads are forcefully suspended by the timer interrupt. With a nonpreemptive scheduler (i.e., cooperative multitasking), threads call the OS when they wish to be suspended (put to sleep.) To illustrate the efficiency of cooperative multitasking, you will add some "cooperation" to the existing preemptive system. The resulting system will be both preemptive and cooperative. A preemption-point is explicitly added to the OS and/or the user software at places where the thread wouldn't mind giving up control of the processor. To immediately suspend a thread, we simply execute

```
TC3=TCNT+15;
```

where the "15" is just big enough to allow the read TCNT, add 15, and store TC3 sequence to complete. You will define a public OS function that puts the running thread to sleep. This function performs just the `TC3=TCNT+15;` operation, nothing more. No changes to the TCB or scheduler are necessary. The tricky part is deciding where in the OS/user software to place calls to this function (preemption-points). One obvious place is in the "spin" portion of `OS_Wait`. Your objective in this part is to keep time-jitter below 30 µs, reduce the number of lost data points, and increase the number of calculations completed, in that order. Rename the file containing the main program so that the S19 file for the first part is available at the time of checkout.

Make a second table showing the three performance parameters for the preemptive/cooperative system versus the size of the two fifos. Again take these measurements when no input occurs in the interpreter.

First Part Checkout (show this to the TA)

1) Run the initial Lab17 software system and explain the logic analyzer trace to the TA,
2) Be prepared to discuss the data you collected as part of the preparation,
3) Run the improved preemptive/cooperative system,
4) Discuss why you placed preemption points where you did.

Description of the Second Part (no preparation is required)

Remove the preemption points from your software, so that it is a pure preemptive system again. The basic idea of the second part is to replace the spinlock semaphores with blocking semaphores. 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 receive data or waiting for there to be room in the transmit buffer. Both types of blocking that will be implemented in the part of this lab. If a thread wishes to output to the display, 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.

All of these features can be implemented by modifying `OS_Wait` and `OS_Signal`. One possible way to implement blocking semaphores is described in Chapter 5 of the book. This implementation uses linked-list data structures to hold the ready and blocked threads. You will need to create multiple blocked linked lists. In general, we will have one blocked list with each blocking semaphore. You will extend the semaphore structure to include both the semaphore value and a pointer to a TCB list containing threads that are blocked on the semaphore. The semaphore initialization should be extended to clear the linked-list of blocked threads on that semaphore. Except for the semaphore structure, everything else in the user program (Lab17.c) should remain exactly the same. In this implementation, a status field and semaphore pointer are added to the TCB. There are other ways to implement blocking, and you are free to implement other blocking schemes.

New `OS_Wait`

1) Save the CCR, then disable interrupts
2) Decrement the semaphore counter, S=S-1
3) If the semaphore counter is less than zero then this thread will be blocked
   set the status of this thread to blocked,
   specify this thread is to be blocked onto the linked list of this semaphore (semaphore pointer)
   suspend thread using `TC3=TCNT+15;`
4) Restore interrupt status

New `ThreadSwitch`

1) Save SP into TCB
2) If this thread is to be blocked
   move the **TCB** of this thread from the active list to the blocked list of the specified semaphore
3) Find the next active thread from the active list
4) Acknowledge C3F, set TC3, and launch next thread

New `OS_Signal`

1) Save the CCR, then disable interrupts
2) Increment the semaphore counter, S=S+1
3) If the semaphore counter is less than or equal to zero then
   wake up one thread from the **TCB** linked list (the one waiting the longest)
   do not suspend execution of the thread that called OS_Signal
   simply move the **TCB** of the "wakeup" thread from the blocked list to the active list
4) Restore interrupt status

You will implement a blocking scheduler. 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, 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 other parts, 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.

Second Part Procedure (no preparation is required)

1) Debug the blocking semaphore implementation.
2) Run the blocking semaphore system for the same FIFO sizes as you used to make the table in first part. Make a third table showing the three performance parameters for the blocking semaphore system versus the size of the two fifos. Again take these measurements when no input occurs in the interpreter. Compare the performance 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. Discuss your results obtained in this part procedure.

Description of the Third Part (no preparation is required)

You should replace the round robin scheduler with a priority-blocking scheduler. If more than one thread with the highest priority is active, the OS will cycle through these active threads and execute each one in turn. The priority order will be `Consumer` (high), `Interpreter`, then `Math` (low). The `Math` thread will never block, and will only run if all the other threads are blocked.

Third Part Procedure

1) Analyze where in the priority-blocking system the largest time-jitter occurs. Make sure the FIFOs are big enough to prevent lost data.

2) Run the priority-blocking semaphore system for the same FIFO sizes as you used to make the other tables. Make a fourth table showing the three performance parameters for the priority-blocking semaphore system versus the size of the two fifos. Again take these measurements when no input occurs in the interpreter. Compare the performance of the priority-blocking semaphore system to the other semaphore systems.

Third Part Checkout (show your listings to the TA)

1) Demonstrate the final system to the TA and discuss your performance data.

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 very 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 on the real 6812. 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) 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. If this is the case, put the complex software into a subroutine and call it from the ISR.
6) *Look for the most recent files on the network.*