Single thread Scheduler
All processes called once each sample

```c
void main(void) {
    init_routines();
    done = 0;
    while (!done) {
        perform_process1(); // Highest priority process
        perform_process2();
        perform_process3(); // Lowest priority
        wait_for_sample_period(); // Waste time here waiting for sample period to expire
    }
}
```

What if the three processes have different sample rates? With a single thread scheduler the following would work but all sample rates would have to be a multiple of the fastest rate.

```c
void main(void) {
    init_routines();
    done = 0;
    samplecnt = 0;
    while (!done) {
        perform_process1(); // Call every sample
        if ((samplecnt%5)==0) perform_process2(); // Call every 5 samples
        if ((samplecnt%12)==0) perform_process3(); // Call every 12 samples
        wait_for_sample_period(); // Waste time here waiting for sample period to expire
        samplecnt = samplecnt+1;
    }
}
```

Here the problem becomes that all processes (in this case all three) need to complete in the time of one sample period. For example at samplecnt = 60 all three processes are called during a single sample period. What if process 3 takes longer than a single sample rate?
int samplecnt = 0 // Global integer

void main(void) {
    init_routines();
    done = 0;
    while (!done) {
        if ((samplecnt%12)==0) {
            //Call every 12 samples
            perform_process3();
        }
    }
}

Hardware Interrupt Scheduler

<table>
<thead>
<tr>
<th>Interrupt Source Vector Table in DSP’s Memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial Port (INT10)</td>
<td>Serial_ISR()</td>
</tr>
<tr>
<td>Timer 0 (INT14)</td>
<td>Timer0_isr()</td>
</tr>
<tr>
<td>Timer 1 (INT15)</td>
<td>Timer1_isr()</td>
</tr>
</tbody>
</table>

// timer 0 interrupts at Sample Period
void timer0_isr(void) {
    perform_process1();
    samplecnt = samplecnt + 1;
}

Could timer0 interrupt timer1 or vice versa? Yes the DSP could be setup to do that but in many cases (and in our case) it is better practice to not allow this to happen. You have to be careful with certain assembly instructions when writing ISR routines to make them “interruptable”.

// timer 1 interrupts at 5 X Sample Period
void timer1_isr(void) {
    perform_process2();
}

The while loop inside main() now becomes the low priority processing loop. Also called the “background” loop. Process 1 and Process 2 have the highest priority. When either timer 0 or timer 1 counts down to 0, the DSP’s hardware automatically stops the current code running in the background loop and jumps to the function specified in the Interrupt Vector Table, in this case timer0_isr() or timer1_isr(). When the DSP is done running the instructions in the corresponding interrupt service routine (ISR), the DSP’s hardware automatically returns and continues processing where it left off in the background loop.

If timer 0 and timer 1 timeout at exactly the same time, timer 0 has the highest priority so its code will run first to completion and then timer 1’s code will be executed. If a timer interrupt occurs while the other timer’s interrupt service routine code is running, the running code continues to completion and then the other timer’s code is executed.
Example Time Load Graph

Units in milliseconds

Process #2 - Highest Priority - Occurs every 2ms - Duration 0.5ms

Process #1 - Occurs every 5ms - Duration 2ms
OMAPL138 SOM from LogicPD

- 300Mhz TMS320C6748 Floating Point DSP
- 300Mhz ARM9 General Purpose Processor.
- 128Mb External RAM/8Mb Flash
- EMAC (Ethernet)
- UART/SPI/I2C/McBSP
- SD Card Interface
- CMOS Camera Interface
DSP/BIOS Example

Arrows in the TSK and between the TSK and HWI Levels indicate direction of multi-thread communication.

User Defined Receive TSK waits for a new character by suspending (or blocking) itself until the semaphore, SEM_UART1RecChar_rdy is set active by HWI #2's function. The new character is then read from the Queue, UART1RecCharQueue. After receiving this character, the task loops back to the beginning of its code and again blocks itself to wait for the next character.

Semaphore: SEM_UART1RecChar_rdy
Queue: UART1RecCharQueue

TSK_UART
Function: uarttsk
- Blocks itself from running until the semaphore, SEM_SendStrmsg_rdy, is set active.
- Fills global array “txbuffer” with the characters given in the Queue, SendStrmsgQueue.
- Enables UART to Send the Characters out at the specified baud rate.
- Blocks itself from running until the Semaphore, SEM_SendStrmsg_done, is set active.
- Loops back to the top to wait for next message to send.

Semaphore: SEM_SendStrmsg_rdy
Queue: SendStrmsgQueue

Global char Array: txbuffer
Semaphore: SEM_SendStrmsg_done

HWI #2
Receives an interrupt from the DSP's UART when there is a new character received in the UART peripheral. This character is placed in a queue and a semaphore is posted signaling the receive TSK to read the character.

Semaphore: SEM_UART1RecChar_rdy
Queue: UART1RecCharQueue

User Created PRDs, SWIs or TSKs call for example SendUART(..) to send a message to the PC or any other UART device (i.e. LADAR). SendUART, places the message in the Queue, SendStrmsgQueue, and the Activates the Semaphore, SEM_SendStrmsg_rdy.

Semaphore: SEM_SendStrmsg_rdy
Queue: SendStrmsgQueue

Global char Array:
Semaphore:
txbuffer
SEM_SendStrmsg_done

HWI #1
This HWI function is setup to send out a new 8 bit character over the UART serial port every time it receives TX ready interrupt from the UART peripheral. When it has sent txbuffer’s number of characters it disables itself and posts the SEM_SENDSTRmsg_done semaphore.

Semaphore: SEM_SENDSTRmsg_done

Arrows in the Hardware Level indicate actual signals and their Direction.

RS-232 standard serial port. (The standard serial port on the back of you PC.)