**Foreground/Background System**

- Most of the actual work is performed in the "foreground" ISRs, with each ISR processing a particular hardware event.
- Main program performs initialization and then enters a "background" loop that waits for interrupts to occur.
- Allows the system to respond to external events with a predictable amount of latency.

```c
unsigned int byte_counter  ;
void Send_Request_For_Data(void)
{
    outportb(CMD_PORT, RQST_DATA_CMD) ;
    byte_counter = 0 ;
}

void interrupt Process_One_Data_Byte(void)
{
    BYTE8 data = inportb(DATA_PORT) ;
    switch (++byte_counter)
    {
        case 1: Process_Temperature(data)  ; break  ;
        case 2: Process_Altitude(data)  ; break  ;
        case 3: Process_Humidity(data)  ; break  ;
        ......  
    }
}
```
ISR with Long Execution Time

Removing the Waiting Loop from the ISR

Kick Starting Output

Interrupt-Driven Output
Preventing Interrupt Overrun

Input Data ➔ Input Ready ➔ Send EOI Command to PIC ➔ Set ISR Busy Flag ➔ Set?

Yes ➔ Ignore this Interrupt!

(Interrupts are re-enabled by the IRET)

Set ISR Busy Flag ➔ STI ➔ Process data, write result to output queue, & kick start.

Clear ISR Busy Flag ➔ IRET

Allow interrupts from lower priority devices and this device too.

When interrupts get re-enabled (see STI below), allow interrupts from lower priority devices (and this device too).

STI ➔ Allow interrupts from higher priority devices.

Send EOI Command to PIC ➔ Disable future interrupts from this device. ➔ Process data, write result to output queue, & kick start.

Clear the mask bit for this device in the 8259 PIC

Enable future interrupts from this device. ➔ IRET

Moving Work into Background

• Move non-time-critical work (such as updating a display) into background task.

• Foreground ISR writes data to queue, then background removes and processes it.

• An alternative to ignoring one or more interrupts as the result of input overrun.

Limitations

• Best possible performance requires moving as much as possible into the background.

• Background becomes collection of queues and associated routines to process the data.

• Optimizes latency of the individual ISRs, but background begs for a managed allocation of processor time.
Multi-Threaded Architecture

- ISR → Queue → Background Thread → Queue → ISR
- ISR → Queue → Background Thread → Queue → ISR

Multi-threaded run-time function library (the real-time kernel)

Thread Design

- Threads usually perform some initialization and then enter an infinite processing loop.
- At the top of the loop, the thread relinquishes the processor while it waits for data to become available, an external event to occur, or a condition to become true.

Thread Design

- Each thread runs as if it had its own CPU separate from those of the other threads.
- Threads are designed, programmed, and behave as if they are the only thread running.
- Partitioning the background into a set of independent threads simplifies each thread, and thus total program complexity.

Each Thread Maintains Its Own Stack and Register Contents

<table>
<thead>
<tr>
<th>Stack</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS:EIP</td>
<td>SS:ESP</td>
</tr>
<tr>
<td>EAX</td>
<td>EBX</td>
</tr>
<tr>
<td>EFlags</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS:EIP</td>
<td>SS:ESP</td>
</tr>
<tr>
<td>EAX</td>
<td>EBX</td>
</tr>
<tr>
<td>EFlags</td>
<td></td>
</tr>
</tbody>
</table>
Concurrency

- Only one thread runs at a time while others are suspended.
- Processor switches from one thread to another so quickly that it appears all threads are running simultaneously. Threads run concurrently.
- Programmer assigns priority to each thread and the scheduler uses this to determine which thread to run next.

Real-Time Kernel

- Threads call a library of run-time routines (known as the real-time kernel) manages resources.
- Kernel provides mechanisms to switch between threads, for coordination, synchronization, communications, and priority.

Context Switching

- Each thread has its own stack and a special region of memory referred to as its context.
- A context switch from thread "A" to thread "B" first saves all CPU registers in context A, and then reloads all CPU registers from context B.
- Since CPU registers includes SS:ESP and CS:EIP, reloading context B reactivates thread B's stack and returns to where it left off when it was last suspended.
Non-Preemptive Multi-Tasking

- Threads call a kernel routine to perform the context switch.
- Thread relinquishes control of processor, thus allowing another thread to run.
- The context switch call is often referred to as a *yield*, and this form of multi-tasking is often referred to as *cooperative* multi-tasking.

- When external event occurs, processor may be executing a thread other than one designed to process the event.
- The first opportunity to execute the needed thread will not occur until current thread reaches next yield.
- When yield does occur, other threads may be scheduled to run first.
- In most cases, this makes it impossible or extremely difficult to predict the maximum response time of non-preemptive multi-tasking systems.

- Programmer must call the yield routine *frequently*, or else system response time may suffer.
- Yields must be inserted in any loop where a thread is waiting for some external condition.
- Yield may also be needed inside other loops that take a long time to complete (such as reading or writing a file), or distributed periodically throughout a lengthy computation.

Context Switching in a Non-Preemptive System

1. Start
2. Thread Initialization
3. Wait?
   - Yes: Yield to other threads
   - No: Data Processing

Scheduler selects highest priority thread that is ready to run. If not the current thread, the current thread is suspended and the new thread resumed.
Preemptive Multi-Tasking

• Hardware interrupts trigger context switch.
• When external event occurs, a hardware ISR is invoked.
• The ISR gets the data from the I/O device and makes a kernel call to enqueue it, causing the state of the thread that is pending on the queue to change from pending to ready. The ISR then calls the scheduler to context switch to the highest priority thread that is ready to run.
• Significantly improves system response time.

Preemptive Multi-Tasking

• Eliminates the programmer's obligation to include explicit calls to the kernel to perform context switches within the various background threads.
• Programmer no longer needs to worry about how frequently the context switch routine is called; it's called only when needed - i.e., in response to external events.

Preemptive Context Switching

Critical Sections

• **Critical section:** A code sequence whose proper execution is based on the assumption that it has exclusive access to the shared resources that it is using during the execution of the sequence.
• Critical sections must be protected against preemption, or else integrity of the computation may be compromised.
Atomic Operations

• Atomic operations are those that execute to completion without *preemption*.

• Critical sections must be made atomic.
  – Disable interrupts for their duration, or
  – Acquire exclusive access to the shared resource through arbitration before entering the critical section and release it on exit.

Threads, ISRs, and Sharing

1. **Between a thread and an ISR:** Data corruption may occur if the thread's critical section is interrupted to execute the ISR.

2. **Between 2 ISRs:** Data corruption may occur if the critical section of one ISR can be interrupted to execute the other ISR.

3. **Between 2 threads:** Data corruption may occur unless execution of their critical sections is coordinated.

Shared Resources

• A similar situation applies to other kinds of shared resources - not just shared *data*.

• Consider two or more threads that want to simultaneously send data to the same (shared) disk, printer, network card, or serial port. If access is not arbitrated so that only one thread uses the resource at a time, the data streams might get mixed together, producing nonsense at the destination.

Uncontrolled Access to a Shared Resource (the Printer)
Protecting Critical Sections

- **Non-preemptive system**: Programmer has explicit control over where and when context switch occurs.
  - Except for ISRs!

- **Preemptive system**: Programmer has no control over the time and place of a context switch.

- **Protection Options**:
  - Disabling interrupts
  - Spin lock
  - mutex
  - semaphore

Disabling Interrupts

- The overhead required to disable (and later re-enable) interrupts is negligible.
  - Good for short critical sections.

- Disabling interrupts during the execution of a long critical section can significantly degrade system response time.

Spin Locks

If the flag is set, another thread is currently using the shared memory and will clear the flag when done.

```
   L1: MOV AL,1
        XCHG [flag],AL
        OR AL,AL
        JNZ L1

   flag = FALSE;
   MOV BYTE [flag],0
```

Spin Locks vs. Semaphores

- Non-preemptive system requires kernel call inside spin lock loop to let other threads run.
- Context-switching during spin lock can be a significant overhead (saving and restoring threads’ registers and stack).
- Semaphores eliminate the context-switch until flag is released.
Kernel suspends this thread if another thread has possession of the semaphore; this thread does not get to run again until the other thread releases the semaphore with a "post" operation.