LAB5, Preemptive Multi-Threading

Goal

Work with pre-emptive kernels. Understand the critical region concept.

Given files

Assuming you copied the lab files according to the lab setup instruction you can get the given files with this command:

```
    cp -r /home/TDDI11/lab/skel/lab5 ~/TDDI11/lab5
```

Your modifications goes to `packet.c`. Use `gmake` to compile.

Background

uC/OS-II is a preemptive real-time multitasking kernel. It is used to build “event-driven” applications in which each thread sleeps (in a “suspended” state) until an external event triggers an associated interrupt. With this kind of kernel (preemptive) the threads do not have to cooperate to get execution time. The kernel make sure that all threads get a chance to execute.

Since the kernel is preemptive, a higher priority thread can suspend an active thread at any point during its execution - forcing it to release the processor. This significantly reduces response time compared to a non-preemptive system (such as Multi-C in previous lab) in which the active thread must explicitly call a kernel function to yield to other threads. In a preemptive system, external events can trigger a context switch by manipulating the registers and stack within an ISR so that it returns to the higher priority thread instead of the one that was interrupted.

As a consequence data inconsistency or corruption may occur. For example if a thread is interrupted in the middle of sending a packet and somone else use the serial line before it get a chance to complete the packet. To prevent corruption data and is often communicated between threads and ISRs using kernel resources such as mailboxes and queues. Shared data or other resources are protected by using kernel mutexes or semaphores.

Assignment 1

This program is similar to the previous, except that in addition to establishing a two-way chat with a display of local elapsed time, each application also displays the remote elapsed time. This requires that each application send and receive both chat text and time text over a single serial line. To distinguish arriving chat text from arriving time text, information is sent as packets in the following format:

```
| 0xff | Type    | N     | Byte 1 | Byte 2 ...... | Byte N |
| Start|0x01=Chat| Byte  |   Actual Data Bytes
| Flag |0x02=Time| Count |
```

One thread assembles incoming packets from a queue filled with data received by the serial ISR. As each packet is assembled, its data is posted to one of two queues according to the type code. Data is removed from these queues by two more threads that update the display.

Implement `SendPacket` function (in `packet.c`). Function prototype:

```
void SendPacket(int type, BYTE8 *bfr, int bytes);
```

Send each byte of the packet by calling the function `SerialPut` (in `serial.c`). Connect two emulated computers (`gmake link`) and test your application. You should be able to send and receive text and both time displays (local and remote) should be running.

Assignment 2

Fix packet corruption problem (in `SendPacket`). If you type very fast, the person at the other computer should notice intermittent corruption of the time displayed in their remote time window (specially when you hit enter). That's because your keyboard interrupts can initiate transmission of a chat packet in the middle of transmitting a (14-byte) time packet from your computer to theirs. To understand consider that `SendPacket(2, “12:34:56.7”, 11)` is called and start sending:

```
0xff 02 11 ’1’ ’2’ ’:’ ’3’ ’4’ ’:’
```

Imagine now that the user start to type on the keyboard and `SendPacket(1, “a”, 1)` is invoked, and the running `SendPacket` preempted. The following is sent:

```
0xff 01 01 ’a’
```

Now `SendPacket(1, “a”, 1)` is completed and `SendPachet(2, ..., 11)` continue sending:

```
’5’ ’6’ ’.’ ’7’ 00
```

The receiving side will now receive the following timer package:

```
0xff 02 11 ’1’ ’2’ ’:’ ’3’ ’4’ ’:’ 0xff 01 01 ’a’ ’5’
```

Followed by the following bytes that have invalid packet header and are discarded:

```
’6’ ’.’ ’7’ 00
```

To solve this problem, you must consider the code in `SendPacket` to be a critical section and protect it against preemption by a semaphore. To do this, you'll need three `mCOS` kernel routines:

`OS_EVENT *OSSemCreate(int count);`

This function allocates and initializes a semaphore data structure and returns a pointer to it. The parameter “count” is set to 1 for a mutex lock (what you need).

`OSSemPend(OS_EVENT *semaphore, int timeout, BYTE8 *err);`

This function returns when it acquires the semaphore; if the semaphore is currently owned by another thread, this function causes the current thread to be suspended while it waits for the semaphore to be released. The parameter “semaphore” is the pointer returned by `OSSemCreate`. Set parameter “time-out” to 0 for no time-out. Parameter “err” is a pointer to a byte that is set to an error code if a time-out occurs.

`OSSemPost(OS_EVENT *semaphore);`

This function causes the current thread to relinquish ownership of the semaphore. The parameter “semaphore” is the pointer returned by `OSSemCreate`.

Assignment 3

Display a count of chat text characters received by the remote computer:

1) Count the number of chat text characters received.
2) Convert the integer count to an ASCII string using the `libepc` function `FormatUnsigned` (see example in `elapsed.c`).
3) Send this string to the remote computer using a packet type code of 3.
4) Modify `ReceivePackets` to handle the new packet type. Use the code for displaying the elapsed time of the remote computer as a guide. This will require creating another queue and another thread.
5) Display the count in a new window located in the middle of the screen.

Deliverables

Code and demo for the lab assistant.