

Timers and event handling

- microkernels often require procedures to be activated at periodic intervals
 - minute, second and hour hand of a graphic clock
 - computer network protocols implementing timeouts
 - implementing a dead mans handle

Implementation of timers and events

- microkernel may require more than one procedure to be timer driven at any one time

- consider a lunar lander monitoring various sensors as it descends
 - altitude reading every second
 - fuel monitoring every 2 seconds
 - oxygen monitoring every 5 seconds
 - general state information transmitted every 10 seconds back to earth

Implementation of timers and events

- not always possible to have multiple hardware timers
 - even if there are multiple hardware timers we might need more...

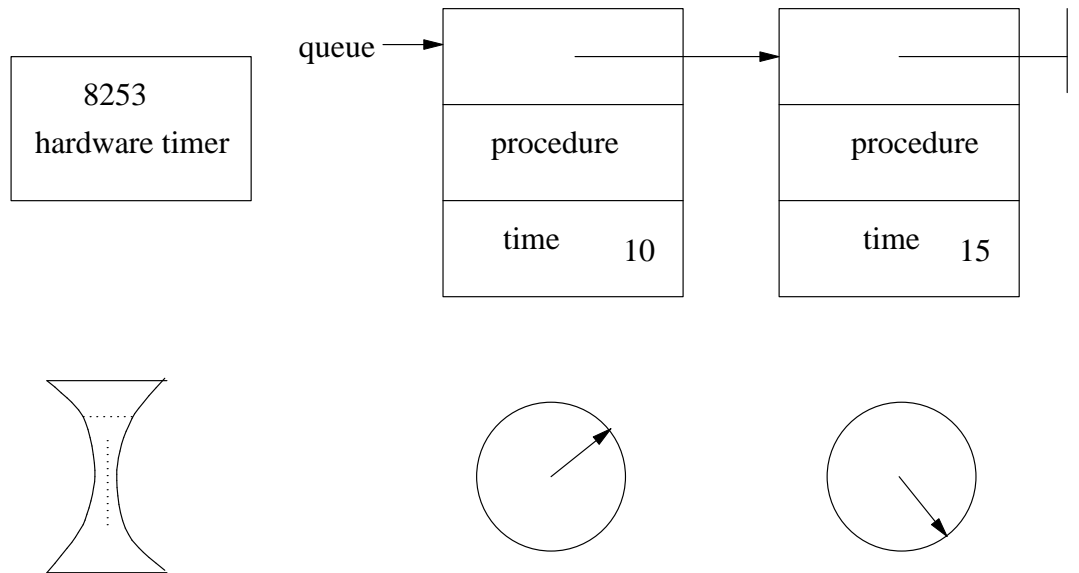
- how can we service many timer procedures with one hardware timer facility?

- most systems will have a hardware timer facility similar to the 8253 as discussed in the earlier part of this course

Solution

- maintain a list of outstanding timer events or procedures which are due to be serviced in the future
 - this list is consumed by a device driver
 - the device driver is activated by the hardware timer interrupt

Solution



- we construct many *virtual timers* through software

Functional interface

- there are many different procedural interfaces which can map onto virtual timers

- these range from
 - activating a procedure in n time units time

- to
 - suspending a process for n time units

- the interested reader should consult
 - Don Libes, Obfuscated C and Other Mysteries October 16, 1992, ISBN-10: 0471578053, ISBN-13: 978-0471578055, Edition: 1
 - Andrew Tanenbaum, Operating Systems Design and Implementation, P155-157, Edition 1, 1987

TimerHandler module

- provides a simple set of timer functions microkernel
- provides the Executive with a basic round robin scheduler
- sets a timer device to issue an interrupt 100 times per second as defined by this constant in TimerHandler.h

```
#define TimerHandler_TicksPerSecond 100  
  
typedef void Event;
```

TimerHandler

- exports an opaque or hidden type, `Event`, upon which all timer actions are held

```
/*  
 * GetTicks - returns the number of ticks since boottime.  
 */  
  
extern int TimerHandler_GetTicks (void);
```


TimerHandler

```
/*  
 * Sleep - suspends the current process for a time, t.  
 *       The time is measured in ticks.  
 */  
  
extern void TimerHandler_Sleep (int t);
```

ArmEvent

- initializes an event, e, to occur at time, t
 - the time, t, is measured in ticks
 - the event is NOT placed onto the event queue

```
extern Event *TimerHandler_ArmEvent (int t);
```

WaitOn

- places event, e , onto the event queue and then the calling process suspends
 - it is resumed up by either the event expiring or the event, e , being cancelled
 - TRUE is returned if the event was cancelled
 - FALSE is returned if the event expires

```
extern int TimerHandler_WaitOn (Event *e);
```

Cancel

- cancels the event, e , on the event queue and makes the appropriate process runnable again
 - TRUE is returned if the event was cancelled
 - FALSE is returned if the event was not found or no process was waiting on this event.

```
extern int TimerHandler_Cancel (Event *e);
```

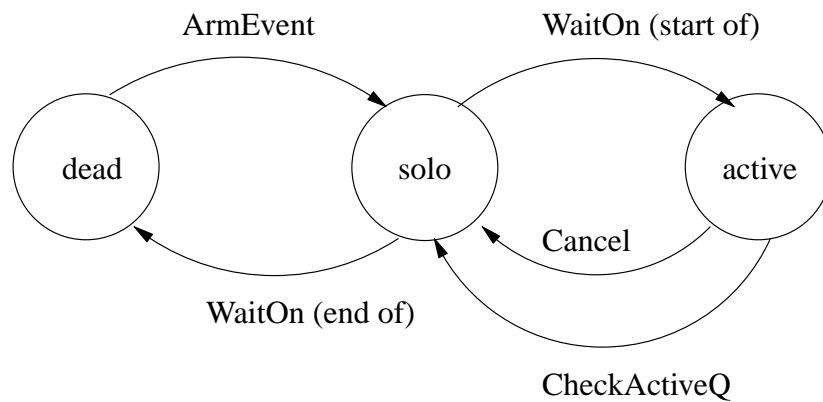
ReArmEvent

- removes an event, e, from the event queue.
 - a new time is given to this event and it is then re-inserted onto the event queue in the correct place
 - TRUE is returned if this occurred
 - FALSE is returned if the event was not found

```
extern int TimerHandler_ReArmEvent (Event *e, int t);
```

TimerHandler.c

- state transition diagram for the events within this module can be described as follows:



KeyboardLEDs Module

- used to switch
 - caps lock, scroll lock and num lock LEDs on and off

- this code turns the scroll LED on

- ```
KeyboardLEDs_SwitchScroll(TRUE);
```

- and this code turns the scroll LED off

- ```
KeyboardLEDs_SwitchScroll(FALSE);
```

How might a microkernel use TimerHandler?

- here is a simple example of use for the TimerHandler
- two processes are created and sleep for 10 and 60 seconds, then print a message
- the process sleeping for 60 seconds can be woken up before the 60 second have expired

TenSeconds

```
void TenSeconds (void)
{
    OnOrOff OldInts = SYSTEM_TurnInterrupts(On);

    while (TRUE) {
        TimerHandler_Sleep(10*TimerHandler_TicksPerSecond);
        Debug_DebugString("..10..");
    }
}
```

SixtySeconds

```
static Event *timeout;

void SixtySeconds (void)
{
    OnOrOff OldInts = SYSTEM_TurnInterrupts(On);

    while (TRUE) {
        timeout = TimerHandler_ArmEvent(60*TicksPerSecond);
        if (TimerHandler_WaitOn(timeout))
            Debug_DebugString("..been cancelled..");
        else
            Debug_DebugString("..60 seconds alarm..");
    }
}
```

Cancelling an event

- is achieved by another process executing this:

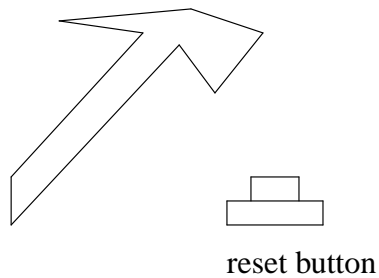
```
if (TimerHandler_Cancel(timeout))  
    Debug_DebugString("have cancelled timeout");  
else  
    Debug_DebugString("not cancelled - expired");
```

How might the timerhandler module be used?

- a watchdog timer is often present in a microkernel
- consider a network router which spans two 802.3 subnets
- or consider some important monitoring software for a chemical plant
 - it must be operational all the time
- admit there *might* be a bug in the system which under some conditions will cause it to hang/crash!

How might the timerhandler module be used?

- one solution is to build a watchdog reset function into the software so that if it crashes the system will reboot



Example: code for hardware watchdog

```
void preventWatchDog (void)
{
    int cancelled; /* not really used */
    Event *e;

    while (TRUE) {
        e = TimerHandler_ArmEvent (halfSecond) ;
        cancelled = TimerHandler_WaitOn(e);
        rechargeWatchDog();
    }
}
```

- requires additional circuitry to allow the system to charge up a capacitor
 - when it discharges it resets the microprocessor

Software watchdog

- debugging technique which allows a program to place messages into a debugging buffer
 - the dead mans handle will activate if no message is dumped within 10 seconds
 - presumably because the system has gone wrong!
 - though not wrong enough to blow away the dead mans handle..

Example: code for dead mans handle

```
static Event *e;

void deadMansHandle (void)
{
    OnOrOff ToOldState = SYSTEM_TurnInterrupts (Off);

    while (TRUE) {
        e = TimerHandler_ArmEvent (10*TimerHandler_TicksPerSecond);
        if (! TimerHandler_WaitOn (e))
            dumpDebugBuffer();
    }
}

void message (char *s)
{
    OnOrOff ToOldState = SYSTEM_TurnInterrupts (Off);

    if (! TimerHandler_ReArmEvent (e, 10*TimerHandler_TicksPerSecond))
        dumpDebugBuffer();
    addMessageToBuffer(s);
    ToOldState = SYSTEM_TurnInterrupts (ToOldState);
}
```