TransSkel
Programmer’s Notes

15: Thread Manager Support

Who to blame: Hans van der Meer, hansm@fwi.uva.nl
Note creation date: 05/15/95
Note revision: 1.00
Last revision date: 07/21/95
TransSkel release: 3.22

This Note describes the implementation of Apple's Thread Manager in TransSkel release 3.22. [The code fragments actually added to TransSkel differ slightly from those described below — Paul DuBois]

1. Thread Manager

The Thread Manager is implemented by Apple from System vs. 7.5 onwards, but for earlier versions of System 7 available as an extension. The version on which this implementation is based is Thread Manager vs. 2.1. This software can be found on the Apple ftp-server. Also available there a draft Inside Macintosh description in Apple DocViewer format (beta draft dated 2/3/95). The coöperative multithread mechanism implemented by the Thread Manager allows the programmer to run several computations (pseudo) concurrently. The most obvious use is the adaptation of programs that run a loop to conclusion, without any regard for intermittent support for an event mechanism. Addition of `YieldToAnyThread()` function calls to these programs very simply turns them into event aware programs.

2. Implementation

Implementation of the Thread Manager is accomplished by changing the inner event loop of TransSkel. The current code in TransSkel.c is:

```
pascal void
SkelEventLoop (void)
{
EventRecord	evt;
Boolean		oldDoneFlag;
long		waitTime;

	oldDoneFlag = doneFlag;		/* save in case this is a recursive call */
	doneFlag = false;		/* set for this call */
	while (!doneFlag)
	{
		if (hasWNE)
		{
		   waitTime = (inForeground ? fgWaitTime : bgWaitTime);
		   (void) WaitNextEvent (eventMask, &evt, waitTime, nil);
		}
		else
		{
		   /* ... */
		   SystemTask ();
		   if (!GetNextEvent (eventMask, &evt))
				evt.what = nullEvent;
		}

		   SkelRouteEvent (&evt);
```

```
        }
        doneFlag = oldDoneFlag;          /* restore in case this was recursive call */
    }
```

This code has been changed to:

```
    pascal void
    SkelEventLoop (void)
    {
    EventRecord  evt;
    Boolean      oldDoneFlag;
    long         waitTime;
    long         see_next_event = 0L;

        oldDoneFlag = doneFlag;          /* save in case this is a recursive call */
        doneFlag = false;                /* set for this call */
        while (!doneFlag)
        {
            if ( TickCount() >= see_next_event )
            {
              if (hasWNE)
              {
                    waitTime = (inForeground ? fgWaitTime : bgWaitTime);
                    (void) WaitNextEvent (eventMask, &evt, waitTime, nil);
              }
              else
              {
                    /* ... */
                    SystemTask ();
                    if (!GetNextEvent (eventMask, &evt))
                          evt.what = nullEvent;
              }

              SkelRouteEvent (&evt);

              /* spend one time quantum in threads */
              if ( hasThreads )
                    see_next_event = TickCount() +
                          (inForeground ? fgTimeQuantum : bgTimeQuantum);
            }

            if (hasThreads)
              YieldToAnyThread();        /* Thread Manager reschedule */
        }

        doneFlag = oldDoneFlag;          /* restore in case this was recursive call */
    }
```

The original event loop gives up control every time it executes `WaitNextEvent()` or `GetNextEvent()`. Such behaviour is a pity for threaded programs, because they have to wait `gWaitTime` or `bgWaitTime` ticks in respectively foreground and background, before their threads get another chance to run. Therefore calling `WaitNextEvent` is deferred until a time quantum of `fgTimeQuantum` or `bgTimeQuantum` ticks, respectively, has been given to the running threads. If these values are chosen sufficiently small, no perceptible degradation in the performance of the system as a a whole will result. And when the threads call `YieldToAnyThread()` often enough, Thread Manager ensures that the event loop is sampled frequently enough. The time quanta can be set to 0, which effectively restores the previous behaviour: calling `WaitNextEvent` every time through the loop.

The flag `hasThreads` signals the presence of Thread Manager. This flag and the other necessary (initialized) variables and definitions have been added to TransSkel.c with statements:

```
/* Thread Gestalt Selectors */
#ifndef gestaltThreadMgrAttr
#define gestaltThreadMgrAttr     'thds'    /* Thread Manager attributes */
enum {
            /* Thread Mgr present */
        gestaltThreadMgrPresent                 = 0,
            /* Thread Mgr supports exact match creation option */
        gestaltSpecificMatchSupport      = 1,
            /* ThreadsLibrary (Native version) has been loaded */
        gestaltThreadsLibraryPresent = 2          };
#endif /* gestaltThreadMgrAttr */

static long fgTimeQuantum = 3L;
static long bgTimeQuantum = 1L;
static Boolean hasThreads = 0;
```

The presence of Thread Manager is figured out in `SkelInit()` by the following code:

```
        /* determine presence of ThreadManager */
        hasThreads = hasGestalt
            && Gestalt (gestaltThreadMgrAttr, &result) == noErr
        #if skelPPC
            && (result & (1 << gestaltThreadsLibraryPresent))
            && (Ptr) NewThread != kUnresolvedSymbolAddress
        #endif
            && (result & (1 << gestaltThreadMgrPresent));
```

A new query selector has been defined in TransSkel.h:

```
# defineskelQHasThreads         12                  /* Thread Manager */
```

To be interrogated through `SkelQuery`, where the switch has been augmented by:

```
        case skelQHasThreads:
            result = hasThreads ? 1 : 0;
            break;
```

The supporting functions `SkelSetWaitTimes` and `SkelGetWaitTimes` have been supplemented by corresponding functions for setting and getting the time quantum values.

In TransSkel.h:

```
pascal void SkelSetTimeQuanta (long fgTime, long bgTime);
pascal void SkelGetTimeQuanta (long *pFgTime, long *pBgTime);
```

In TransSkel.c:

```
pascal void
SkelSetTimeQuanta (long fgTime, long bgTime)
```

```
{
	fgTimeQuantum = fgTime;
	bgTimeQuantum = bgTime;
}

pascal void
SkelGetTimeQuanta (long *pFgTime, long *pBgTime)
{
	if (pFgTime != (long) nil)
		*pFgTime = fgTimeQuantum;
	if (pBgTime != (long) nil)
		*pBgTime = bgTimeQuantum;
}
```