Threads

by Frederick Hiltz, Stage Manager

"Threads" has become a hot topic now that popular desktop environments support this useful programming technique. Long available in many operating systems and in the M language, the rediscovered method deserves a place in every expert programmer's tool box. Properly used, threads can increase speed, simplify the construction of a program, and enhance the user's perception of its execution. Threads are separate paths through one program that *execute concurrently*, *share variables*, and *synchronize* their operations.

The operating system literature discusses multithreading and multitasking with esoteric terms like *lightweight process* and *mutex*. Do not let these distract you; our familiar M has what it takes.

Concurrent execution: The JOB command starts a new thread. Most implementations do this quickly enough for the new thread to contribute without noticeable delay.

Shared variables: Globals are so familiar that we forget how strange this concept is in other languages.

Synchronization: A global variable or a LOCK command synchronizes the threads.

The most common way to assemble these pieces into a multithreaded program is the boss/worker model. The boss thread controls everything, dispatching worker threads for specific tasks and coordinating their results. For example:

- A word processor runs its spelling checker after the typist enters each word. In its own thread, the spelling checker does not slow the boss thread, which learns after each key stroke whether the spelling checker found an error.

- While a nurse enters a medication order, the program compares it with the patient's allergies to medications. The allergy check presents its warning before the order is completed.

- Several computers execute pieces of a very large computation.

Another multithreading architecture is the producer/consumer model. Neither thread controls the other, but one produces a resource while another consumes it, and neither need wait for the other. For example:

- A radio page function takes pager messages from calling programs, places them in shared variables, and returns immediately. A consumer thread sends the messages when the radio transmitter becomes available; it marks them "sent" in the shared variables, where the producing function can check their status.

- A language processing program passes text through three stages that identify words, parse syntax, and analyze semantics. In separate threads the three stages run faster on a multiprocessor computer.

When should one consider multithreading? If an interactive program displays "Please wait for ..." (or ought to), then look for a chance to move the obstruction into a separate thread. Does a task have two long parts in sequence that could run in parallel? Perhaps one thread can continue while another blocks waiting for a resource. A compute-bound process and an I/O-bound process can overlap successfully on a single processor, and multiple processors can do wonders for intensive computations. Finally, do not overlook the simplicity of a program constructed from single-purpose threads.

There's no such thing as a free lunch, of course. Because performance often prompts the choice of multithreading, some experiments may be useful to measure the costs and benefits in your M environment. Here are some topics to consider:

Concurrent execution: Is the JOB command fast enough? Plan a strategy for when the implementation cannot start any more jobs. Provide a way for the threads to identify each other.

Shared variables: Arguments of the JOB command are fast but limited. Global variables are surprisingly fast when jobs share their memory buffers. Some implementations offer job-to-job communication through shared memory.

Synchronization: Here lie most of the interesting parts of multithreading. Common background tasks such as filing and print spooling work alone, but threads must coordinate their work. Sometimes a global "done flag" set by a worker and examined by a boss is adequate, but a robust program usually requires more. Depending on the application, a worker might report progress, several different failures, and success to the boss. What if the worker never starts? What if it stalls before reporting? How long should a consumer wait for a producer to produce? You can program these with timed locks guarding global status variables of any complexity.

For example, a boss thread starts three worker threads named 1, 2, and 3. Each worker locks ^ALEX(name). When the boss is ready for results, it executes LOCK ^ALEX::10, which succeeds when all three workers have halted and placed their statuses in a shared variable, or fails when any worker takes an unreasonable time.

This example does not account for the worker that never starts and thus never locks its ^ALEX(name). Can you devise a synchronization method that informs the boss if this occurs? Note that a tight loop checking a global variable (spinning) is bad form—it burns resources needed by the workers. Try your hand; this might be your first step into the world of threads.

Reference

A. D. Birrell, *An Introduction to Programming with Threads*, Systems Research Center, Digital Equipment Corp., Palo Alto, Calif., 1989.

Frederick L. Hiltz, Ph.D., develops medical information system software at Brigham and Women's Hospital, Boston, Massachusetts. His E-mail address is: fhiltz@bics.bwh.harvard.edu

---

Do you have a question that deserves discussion? Have you found a good answer to someone else's question that you would like to share? How about a controversial question and a discussion of pros and cons? If you prefer that your name not be published, please say so in your contribution, which should be sent to the Managing Editor at *M Computing*.