COMP30640 Report

Subject: Threads - *A System for the Support of Concurrent Programming*

The paper I've chosen describes the idea behind, and implementation of, the Threads system - described as a lightweight alternative to processes for parallel computing and multiprogramming.

Problem Description

Concurrent programming is an important aspect of modern applications. For example, intensive numerical simulation of dynamical systems might be spread across hundreds or thousands of cores in a supercomputer, or a server might handle incoming requests from many users around the world simultaneously. Much closer to home, day-to-day working applications like Firefox or Microsoft Outlook also use concurrent programming in a real time setting - Outlook might be at once updating the GUI for drawing, sending and receiving messages, analysing text, and checking for updates. Firefox might be playing audio from multiple tabs while displaying the current tab, while downloading a file in the background.

The requirement that a computer do multiple things at once was far from new at the time the paper was written - operating systems have similar needs, especially in the real time case - responding to the terminal, running multiple applications, doing jobs in the background. Processes are important here - the OS might have multiple (perhaps hundreds) of processes running at once and manage them so that (hopefully) each gets access to the CPU, memory, and other resources they need to complete in a timely fashion.

The difference here is that it is a single application that is performing concurrent programming - any single GUI application will likely have tasks that need to be done quickly and continuously, like updating buttons and progress bars, and tasks that take a long time to perform, like copying files from one location to another. This can be done using processes - a new process could be created to perform the copy operation. This has the disadvantage of being expensive - for a long copy operation, this might not matter, but if the processes were to be started for a 20ms computation the overhead might be significant[[1]](#footnote-1). Furthermore, if the progress bar is to be updated continuously this would require continuous inter-process communication, which is expensive[[2]](#footnote-2).

The main reason for this expense is that processes do much more than allow for concurrent programming. They provide protection to processes from other processes, for example. Hence the requirement that communication between processes pass via system calls. There is no need for this kind of protection in most[[3]](#footnote-3) in-application concurrent programming needs - a GUI for example is expected to have access to at least the state of that which it is displaying and allowing interaction with.

Another approach to the GUI example would be to do it synchronously, with careful setup. Ultimately, a single processor computer is doing everything "synchronously" at some level. In this case, it would amount to performing to performing the long task in small chunks and checking GUI events regularly. At this point, the programmer is setting up its own internal scheduling scheme for two tasks. This would get tedious and tricky and error-prone if the application were filled with handcrafted scheduling schemes. Ultimately, it is this approach that Threads seek to generalize and, most importantly, make easily accessible to the programmer. The programmer should only have to specify where threads are created, what they do, and where they join, at a high level. The scheduling, multiprocessor management, interrupt and exception handling and other implementation details are handled by the "threading" library.

Comparison to Related Works

This paper is not the first to suggest threads as a concept. It mentions other similar works. The Mach kernel may[[4]](#footnote-4) have been the first to include threading. Note that some of the thread implementation in Mach is at least partially within the kernel. The functionality given by Mach is low level, and the intention was to have libraries make a bridge between it and the end-programmer - one such library was C-threads, also mentioned in the paper this report is about. Like the Threads system, C-threads offers synchronization mechanisms and support for exception handling/interrupts. The author of the Threads system describes their system as experimental, but structured in a clear and extensible manner, when compared to other libraries.

I don't know anything about how c-threads is structured internally, or if it is extensible/how easy it is to extend, however.

Solution

Threads is described as a library of procedures, which run in user mode, to help the current program. Note that at this point the threading scheme belongs to the application - the OS does not have to be aware of it at all. The library is designed on an approach similar to object-oriented programming[[5]](#footnote-5), allowing it to be extended by users.

The library is split into two levels - the second level makes use of procedures from the first, "bottom" level, while the user is intended to use the second level. The user can add more levels if they require. In order for the procedures to work as expected the user must be careful to use them in their intended order/context, since there is no form of built-in protection to alert of or prevent their misuse.

Bottom Level

The first level deals with context-switching, interrupt/exception handling, stack allocation, and some synchronization.

Perhaps the most important procedure here is "run()", which looks at the current queues, makes a scheduling decision, and chooses to run one of the threads. When run switches the running thread, it does something similar to the "context switch" an OS does when switching between processes - the state of the current running thread is saved to a data structure, and the state of the next thread is loaded into the processor.

There are also procedures for setting code to run when an exception is raised, or an interrupt occurs. This allows for an exception to handled in a new thread, for example.

There is a part of the bottom level dedicated to stack frames. I didn't fully understand this part, but my (tenuous) current idea is that this part allocates memory for local variables and return locations for each thread. When an operating system allocates a stack for a process, the process usually assumes that the stack can grow as much as is needed. The OS provides an abstract logical address space and makes sure that addresses as they are needed are mapped to real locations in memory. A similarly growing address space for each thread would be helpful, however, for simplicity, the authors stuck to a system where the stacks are fixed from the beginning.

Top Level

The top level is described as the "runtime library" of the system - the user will mostly be interacting with this in their program (or when they are adding additional layers).

One important new construct in this level is the exception safe monitor - a synchronization method where threads enter queues waiting for access to a certain resource. Exception safety means a record of what monitors a thread has currently entered is needed, so that when an exception occurs, and thread execution stops, the relevant resources can be released to other threads.

Another feature is IO handling - the library offers so called "synchronous" IO calls which do block the current thread but do not block the process as a whole. Other threads are allowed to continue. (As far as the OS is concerned these are asynchronous calls made by the process)

Finally, there are different ways of handling interrupts - a new thread can be created for the interrupt, a specific existing one can be paused, or an exception can be raised.

Scientific Method and Results

There is not much to say here - the paper is more like introductory documentation, or maybe a development progress report, than a scientific paper. It describes their idea and implementation, and very briefly provides some current performance figures.

The Threads system as a whole seems a practical exploration into how to build a threading library, with the hope of at least building a foundation for future works. When viewed like this, the research question could be "How do we build a system to help leverage concurrency on an in-application scale?", and the results are, as of when this paper was written, positive, but ongoing.

Personal Impressions

Threads feel like an invaluable step towards easy in-application concurrent programming. I've had some experience with them through Java Swing and C# WPF - the typical need of a GUI application needing to do background things without becoming unresponsive. Here, concurrent programming makes things far easier.

C# offers an additional layer of abstraction from threads in the form of "async" methods. Calling an async method, and "awaiting" on it later, can implicitly indicate the creation of a thread. The Java 8 Stream API had an interesting hook into threading as well. It's possible to create "foreach" loops marked as capable of being executed in parallel (each iteration independent of the others), for example, and this too might implicitly cause the creation of threads for concurrency.

I don't have much experience with either of those[[6]](#footnote-6), but it looks like we might end up with almost entirely implicit concurrent programming being the norm someday, with only minimal, concise hints to the compiler as to real-time/synchronization constraints where there is ambiguity.

With regards to the implementation described by the paper itself, their description was succinct and written in language easy for a reader with only introductory computer-science studied so far to approach. Their brief description of object-oriented programming as a message base system fascinated me, as did the referenced paper that demonstrated a form of "duality" between procedures and OOP. I don't know if OOP has to this day a different meaning in the field of operating systems, or if our concept of OOP has evolved significantly since then. I guessing it is a little of both.

On a related note, I was surprised that the fundamental idea of a "Thread" does not seem to have changed much since this paper was written.

In terms of limitations of the paper, the first one that comes to mind is lack of detailed information regarding how successful this iteration of threading is. I mentioned this before in "Scientific Method and Results". There is also the limitation they describe in that shortcuts were taken for the sake of making a prototype quickly - the fixed stack allocations, and the fact that the multiprocessor version is unoptimized especially in comparison to the single processor version.

1. I'm not sure about the exact overhead, but it looked like creating a process for a bash script was taking between 100 microseconds and a millisecond on my system. [↑](#footnote-ref-1)
2. I also found it to be more awkward to code on the one occasion I needed it in C#, though I imagine that is more to do with the prevalence of threads today and my lack of practice than any inherent limitation of processes - this could all be abstracted away in any case. [↑](#footnote-ref-2)
3. Browser tabs are a good debatable counterexample to this. An application might actually want to leverage the extra security provided by processes. Especially between components of the application that shouldn't be communicating much anyways. [↑](#footnote-ref-3)
4. I'm choosing not to dive down the full history of early operating systems, but Mach frequently comes up as an example first implementation. [↑](#footnote-ref-4)
5. It is very similar in how procedures are applied to data structures as actions and how these procedures and data structures can be extended, however, I'm not sure if it adheres to the encapsulation/protection principle of OOP. [↑](#footnote-ref-5)
6. And I'm not entirely sure I've described their capabilities accurately, having only made token experiments with them. [↑](#footnote-ref-6)