Sam Britt (901-75-9808)
Yohanes Suliantoro (902-76-1560)

The (almost) bug-free GTThread project

The overall structure of the new version of `gtthreads` had two main goals: (a) to improve the clarity and robustness in the use of `setjump` and `longjmp`, and (b) factor out all the scheduler-specific code into modules that adhere to a generic "scheduler" interface, with hooks put in place throughout the `kthread` and `uthread` modules to initiate scheduler-specific functionality.

Context Switching

The previous version of `gtthreads` relied on the Unix functions `sigsetjmp` and `siglongjmp` to save and restore the user space context (that is, the values of all the registers, including the stack pointer, and the program counter). Though this is a common technique, these (and related functions) can make programs hard to read and understand because control flows into and out of the middle of functions (they are acting as very powerful goto's). Therefore it is imperative that the use of these types of functions be minimized only to where it is necessary, and and that their use is clear. With that in mind, we noticed some issues with the implementation.

First, context buffers were used in both the kthread and the uthread implementation. We found that the uses of setjmp and longjmp for the kthreads not to be necessary, and were able to remove those jumps completely. In the original version, before uthreads are created each kthread was held in a loop consisting of assembler to yield the processor, a setjmp, a function call, and then a longjmp back to the beginning of the loop. We removed these jumps and replaced them by a single loop that, until an exit condition is made simply sleeps using `nanosleep()`. Signals are used to notify the kthread to schedule a new uthread. The `nanosleep` function is used instead of the standard `sleep()` because `nanosleep` is guaranteed not to interfere with the alarm signals used by the timers (which are in turn used by the schedulers).

Second, the design of the package was such that the "main" kthread---that is, the kthread

that runs the user-facing `gt_app_init()`, `gt_uthread_create()`, and `gt_app_exit()` code, is somehow special. If there are *n* processors on the system, *n* kthread's will be used as lightweight processes to schedule all the uthreads that the user creates. However, in `gt_app_init()`, only *n* - 1 kthreads are cloned (and enter into the setjmp/longjmp loop described above), because the first kthread is the one actually running `gt_app_init()`, and will return to the user. This main kthread has a completely different execution path from the others---it will return to the user, spawn all the uthreads (potentially putting its sibling kthreads to work, but leaving its own uthreads idling in its ready queue), and execute `gt_app_exit()` at the user's request. It is not until `gt_app_exit()` that the main kthread begins to allow its uthreads to execute. This design has **dire** consequences which are not obvious at first. Essentially, if execution for the main kthread is delayed (by either intense computation by the user before uthreads are created, or after all the uthreads are done but before `gt_app_exit()` is called), the main kthread can potentially get interruped by the scheduler timer and execute a longjmp before ever issuing a setjmp, which is undefined behavior. The original design assumed that the main kthread would create all the uthreads and execute `gt_app_exit()` within a uthread timeslice.

Our design solves this issue by having, for *n* processor, *n* identical kthreads to run the user code, and an additional main thread that only issues the user-facing `gt_app_init()`, `gt_uthread_create()`, and `gt_app_exit()` code. This special kthread simply loops in `gt_app_exit()`, waiting for the other kthreads to finish. Being completely separate, there are no special cases in signal handling, execution flow, etc.

Finally, setjmp and longjmp themselves can be replaced by the `getcontext`, `setcontext`, `makecontext`, and `swapcontext` suite of functions. These are slightly less portable, but they should be supported on any modern Linux distribution, our target platform. The offer much the same functionality as the *jmp functions, but the interface is a bit easier to use and understand. Notably, the older version of this package was forced to to use some trickery involving special signals, `sigaltstack()`, and a substantial amount of logic to support alternate stacks for the uthreads. By using makecontext(), the stack can simply be allocated and assigned, as these are supported by the interface.

Scheduler Encapsulation

By encapsulating the scheduler-specific code into modules that follow a generic interface, the user can provide the scheduler she wishes to use as an option to `gtthread_app_init()`, and the scheduling policies can be loaded at runtime. This also allows the programmer to better understanding of the program flow and ensure the semantics of the program are correct. The difficulty in isolating individual component functionality makes it hard to debug each component separately (unit testing). It also makes it difficult to repeat tests. While this process has fixed some bugs, it may also have introduced new bugs.

Both the priority scheduler provided with the package and the new completely fair scheduler were designed to meet this generic interface. The interface includes the functions:

- `scheduler_init()` This function is called at the start of the application, and allows the scheduler to initialize any data structures it needs. The scheduler will have access to its data through a global variable.
- `kthread_init()` Called after the initialization of every |kthread|, to allow the scheduler to do any bookkeeping it may need to do for each `kthread`
- `uthread_init()` Called after the creation of every uthread. The scheduler must return an appropriate kthread for it to be scheduled on.
- `preempt_current_uthread()` Called after the timer interrupt to preempt the currently running uthread. It is here that the scheduler can insert the uthread into an appropriate ready queue, if needed.
- `pick_next_uthread()` Called to choose the next runnable `uthread`. The scheduler can implement its priority policies here.
- `resume_uthread()` Called just before returning control back to the user's application. Its main function is to set a timer for the appropriate timeslice amount. The library will be woken at this point to schedule the next thread.

All these functions are set up through function pointers maintained in a global `scheduler` data structure. The pointers are defined at application startup, so there should be little or no performance lost during the scheduling of uthreads.

For the completely fair scheduler, the data structures are as follows: The scheduler maintains an array of `cfs_kthread` structures, one for each virtual processor being used. Each `cfs_kthread` maintains a pointer to the `kthread` structure used by the rest of the system,

as well as CFS-specific data. For example, each `cfs_kthread` has a pointer to the red-black tree holding all the schedulable entities for that processor, a pointer to the currently running `uthread`, the current latency or “epoch length” for the system, the current load on the system, and the current value of the minimum virtual runtime (`vruntime`) of all the uthreads in its runqueue. Similarly, the `cfs_uthread` structure maintains a pointer to the corresponding `uthread` used by the rest of the system, in addition to the current values of it's virtual runtime, its priority, and a pointer to its node in the red-black tree.

At each point of the above interface, these data structures are updated appropriately. For example, during `uthread_init()`, a node is created with the current minimum vruntime and inserted into the appropriate `cfs_kthread`’s red-black tree. In addition, the cfs_kthread’s load is increased, and this potentially increases the latency past a threshold. At `preempt_current_uthread()` the uthread’s vruntime is updated appropriately and inserted back into the red-black tree. The uthread with the minimum vruntime is chosen at `pick_next_uthread()`, and its timeslice is calculated at `resume_uthread()`. Finally, when a `uthread` has completed execution, it is removed from the red-black tree and the kthread load is reduced.

The compiled in default is a latency of 40 ms. Virtual processors are chosen in a round-robin fashion; there is no co-scheduling or grouping of uthreads. Times are to microsecond resolution. The red-black tree implementation used can be found at http://www.mit.edu/~emin/source_code/red_black_tree/index.html. It was modified to support caching of the left-most node for quick retrieval.

Outstanding Issues

Currently, the package works with 128 threads, with matrix of size 32, 64, 128, 256 or 64, 128, 256, 512. Occasionally there may be segfault error, but it is quite rare compared to the original package. However, if the package is run with debug parameters, it may often result in segfault or infinite loop due to a missing kthread. This problem may be caused by incorrect timing caused by large number of print out to the console. Future work for this package will need to address this issue, which is quite likely due to missing error checking.

The package is heavily reliant on signal handling, thus there could be potential problems with

missing/ uncaught signals, signals interfering with each other, and the use of async-unsafe system calls. This setup may make future work more difficult to debug. Another issue that needs to be addressed is synchronization. While the gt_spinlock has locking mechanism, it does not provide a way to do condition variable atomically. Additional test using pthread mutexes and condition variables, or perhaps POSIX semephores, to replace current gt_spinlock implementation might provide better base code for the package. These synchronization mechanism will make future work on the package more stable.