Converting Single-threaded Code to Multi-threaded Code

Converting a single-threaded codebase into a multi-threaded one is not a simple task. Certain rules must be followed to ensure a successful transformation. A thread generally consists of multiple procedures that may include local variables, global variables, and parameters. Global variables for one thread may not be global across the entire program; they are considered global because they are used by multiple threads, but other threads may only have read-only access to them.

Example: Issue with Shared `errno` Variable In Unix, when a process or thread makes a system call that fails, the error code is stored in the `errno` variable. Suppose Thread 1 requests access to a file, and the response is stored in `errno`. Before Thread 1 can read `errno`, its execution time ends, and it is suspended. Thread 2 is then executed, makes a system call that also fails, and the error is recorded in `errno`, causing it to be blocked. When Thread 1 is reloaded and reads `errno`, it finds the wrong value and exhibits incorrect behavior.

[[Pasted image 20241016225511.png]]

Solutions to Global Variable Conflicts

Several solutions exist to resolve this issue. The simplest approach is to prohibit global variables entirely. However, this can lead to many conflicts. Another approach is to give each thread its own global variable. In this case, each thread has its own version of `errno`, preventing conflicts. This creates a new level of variable visibility, where variables are visible to all procedures within the thread.

[[Pasted image 20241016225541.png]]

This solution is not always easy to implement, as most programming languages do not natively support this additional level of variable scope. It is possible to reserve a portion of memory for global variables and pass it as a parameter to the thread procedures.

More Complex Solution: Custom Library Procedures

A more complex solution involves adding new library procedures to create, define, and read thread-local global variables:

- **Create_global("bufptr")**: Allocates memory for the pointer `bufptr` in a memory segment or a storage area reserved for the thread. Only the calling thread knows the location of its global variable. If another thread creates a variable with the same name, it will be assigned a different memory location, avoiding overwriting.

Example: Thread-local Global Variables In Unix, as described earlier, when a thread makes a system call, the error code is stored in `errno`. If Thread 1 is suspended and Thread 2 updates `errno`, conflicts arise. The solution of using thread-local global variables can prevent this issue.

[[Pasted image 20241016225748.png]]

To access these variables, two system calls are needed for reading and writing:

- **Set_global("bufptr", &buf)**: Stores the value of a pointer at the location created by `Create_global`.
- **bufptr = read_global("bufptr")**: Returns the address stored in the global variable, allowing access to the saved data.

[[Pasted image 20241016225815.png]]

Non-reentrant Library Procedures

Another problem arises during the transformation to multi-threading: many library procedures are not reentrant, meaning they cannot handle multiple calls simultaneously. For example, in Unix, the memory allocation procedure `malloc` maintains essential tables for memory usage. While `malloc` updates these lists, some pointers may not point to valid memory locations. If a thread switch occurs at this moment, and the new thread uses an invalid pointer, it may cause the program to crash.

A solution is to use a **wrapper code**, which adds a flag to indicate whether the library is in use. If another thread tries to access the same library, it will be blocked until the current thread completes its operation.

Signaling Issues in Multi-threading

During execution, threads may use signals, which can also pose problems during the transition to multi-threading. If a thread calls `alarm`, it is crucial that the result is sent to the requesting thread. However, the kernel does not recognize individual threads and cannot directly communicate the response. Additionally, if a process has only one alarm and multiple threads make alarm calls simultaneously, conflicts can arise.

Stack Management in Multi-threading

Another issue is stack management. Normally, when the kernel detects that a process's stack has overflowed, it automatically increases its size. However, in a multi-threaded environment, the kernel does not recognize stack overflows per thread, resulting in an unresolvable error from the kernel's perspective.

Conclusion: Challenges in Converting Single-threaded to Multi-threaded Systems

Many challenges can arise when transitioning from single-threaded to multi-threaded code, which raises compatibility issues. Although these problems are solvable, the complexity shows that transforming an existing system into a multi-threaded one is not simple. It requires rethinking the entire system to ensure compatibility with existing programs.