Dead ways in multithreaded programming

Zdeněk Kotala
Revenue Product Engineer
Sun Microsystems
Agenda

• Process life
• Signals
• Atfork handlers
• Memory access
• Sessions
Why

• Nine years ago I spent one month to develop multithreaded application and three months to hunt a bugs.
• In former company I spent 3 weeks to rewrote signal handling to work correctly.
• Now because 6842872, 6828366, 6823591, 6548350, 6276483, ...
Process life - I

• Before main() function compiler and linker add prologue which setup libraries.
• After main() compiler and linker add epilogue which call exit(2) from libc.
• Exit(2) function call all atexit handler and close all opened files. After that it calls _exit(2) syscall.
• _exit(2) syscall start to cleanup process and their threads.
Process life - II

• Prologue runs as single threaded and .init sections are processed in the main thread, but if library is dynamically opened by dlopen(3C), application can have more threads already.

• When exit(2) is invoked other threads are still active and running. Clean up (e.g. atexit handlers, .fini) usually causes fatal errors or crash.
Process life - III

- Do not except that .init section runs in single threaded process.
- Do not call exit(2) function when more threads are running.
- Dedicate one thread (usually main thread) which control worker threads and which is responsible for cleanup.
Signals - I

• Signal is asynchronous event used for inter process communication.
• When signal arrives and it is not blocked one thread is interrupted and the thread runs signal handler.
• Signal handler runs in parallel with other threads.
• Each thread has own signal mask which is inherited from parent thread.
• Because list of signal safe functions is limited dedicate one thread to signal processing is better.
• DO NOT use mutexes in signal handler.
Signals - II

void *sigint(void *arg)
{
    int sig;
    for(;;)
    {
        sigwait ( &signalSet, &sig );
        if ( sig == SIGINT )
        {
            printf("Got signal SIGINT\n");
            return NULL;
        }
    }
}
Atfork handlers I

- Fork(2) calls create children process which inherits only calling thread, but all mutexes, condvariables and so on stay in state before fork(2). For example some mutexes can be locked.
- It is important (especially for library) to handle it correctly.
- pthread_atfork(3C) allows to setup handlers which are called before and after fork. Handlers should acquire all mutexes before fork and released it after.
- Atfork handlers are processed in parallel with other threads.
Atfork handlers II

• Atfork handler have to be setup before any lock is acquired.
• Order of handlers registration is important. Wrong order can lead to deadlocks.
• Fork(2) and pthread_atfork(3C) use internally same mutex for atfork handler list access.
Atfork handler III

Note: Linking application directly against to pkcs11_softtoken and pkcs11_kernel is not recommended.
Atfork handler IV

Thread 1

C_Initialize(...)
    pthread_mutex_lock(&global)
    ...
    pkcs11_slot_mapping(...)
    dlopen(softtoken)
    .init
    pthread_atfork(...)
    pthread_mutex_lock(&atfork_list)

Thread 2

...  
fork()
    pthread_mutex_lock(&atfork_list)
    pkcs11_atfork_prepare()
    pthread_mutex_lock(&global)
    pthread_mutex_lock(&atfork_list)
Parallel memory access

- Access to shared memory has to be protected by lock. It is not necessary only in few cases.
- Locking is expensive and also critical section length has impact on performance and scalability.
- Using one giant lock is easy to implement, but application scalability is poor.
- Locking has to be designed at begging of development. Any future lock splitting is expensive and it is root cause of many bugs.
- Prefer pthread_rwlock for better scalability.
atomic.h

- Solaris offers a bunch of atomic operations in atomic.h (atomic_ops(3C), membar_ops(3C))
- It is good when we need simple data structure modifications.
- Unfortunately, functions are not portable.
- Membar_ops are generic memory barriers which are dedicated to guarantees read/write memory order.
Thread Local Storage (TLS)

• Thread local storage is a method to store thread-specific data.
• POSIX defines `pthread_key_defines`, `pthread_setspecific`, `pthread_getspecific` function.
• Compilers offer syntactic sugar. For example:
  ```
  __thread int localint;
  ```
Sessions

• Sessions are used to keep state information of communication between client and server or application and libraries.

• One session should not be used in different threads. Parallel usage causes usually crash or strange behavior.

• Session pooling is used for resource reduction in some cases, but usually it has limitations.
Thanks to

Julius Štroffek
Chris Beal
guug
Dead ways in multithreaded programing

Zdeněk Kotala
zdenek.kotala@sun.com