Thread: Attributes

Sonntag, 25. Dezember 2022 09:38

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Disclaimer

This article is not about the concurrency introduced with C++11 along with the new memory model (these two came hand by hand).
It's more a special "*side dish*" to that - for customizing it to the embedded (*Linux*) domain, where additional *POSIX-like* attributes and CPU affinities need to be taken into consideration, in order to gain the best performance - especially for the realtime threads.

The source code is just a wrapper around the *std::thread*, as a starting point, extended with ability to set these extra features.
For C++20 and beyond, one should use *std::jthread* instead, with built-in cooperative cancelation mechanism.
https://github.com/josuttis/jthread

The source code: https://github.com/damirlj/modern_cpp_tutorials/blob/main/src/Thread/ThreadWrapper.h

Scheduler/Priority

Normal scheduling

Default scheduling policy in Linux is **SCHED_OTHER** (POSIX: SCHED_NORMAL).
This is also known as **CFS- Completely Fair Scheduler**, where the kernel will specify the equal time slices for serving the threads by CPUs as computing resource. The all threads share the same - **static priority 0**.
The way to distinguish the threads within the same priority group is through the *niceness* - as dynamic priority that will be used as weighting factor for that initially equal execution interval.
The verbal rule is: *"The more **nicer** thread is, the **less** priority it has"*.
The range of niceness is **[-20, 19]**.
To translate it to the priority scale, simple formula can be applied: PRIO = 20 - NICE, which gives us a range: [1 - 40].
The all resource limitations imposed by the kernel (S-Software, H-Hardware), one can check in terminal with

```
# ulimit -Sa
```

or just for the niceness max range

```
# ulimit -e
```

Linux provides *setrlimit* for setting - overriding these constraints at software level, where the hardware limitations can't be exceeded (can be through the software limitations only lowered, and therefore turned into more restrictive ones).
Privileged processes with CAP_SYS_RESOURCE capability can make arbitrary changes.

→ **Digression**
Capabilities are subset of privileges of a root process, that can be individually assigned to the non-privileged processes (actually, capabilities are per-thread credentials), which determine the type of the system API they can exercise: call, in order to access and modify the system resources (including the thread scheduling/priority: CAP_SYS_NICE).
You can check it (different capability sets) with

```
# cat /proc/<pid>/status | grep Cap
```

or for individual threads of a process

```
# cat /proc/<pid>/task/<tid>/status | grep Cap
```

Pay attention to the *Bounding* set, since this is a bit-mask that will be AND to the all other capability sets.
It's a superset of all capabilities that process can gain during an *exec*() call.
Effective set is the one which kernel actually uses for checking the permissions, and it's subset of the *Permitted* set - removing the capability from this set is irreversible.
@note If the CAP_SETPCAP is not in the *Bounding* set - you can't *add/remove* capabilities

- either programmatically (https://linux.die.net/man/3/cap_set_proc)
  - *Add* - raises capability from *Permitted* to *Effective* set
  - *Remove* - clear capability only from *Effective* set, without affecting the *Permitted* one
- nor using *setcap* command on the running binary, to do it on the fly.

https://man7.org/linux/man-pages/man7/sched.7.html
https://linux.die.net/man/2/setrlimit
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/33528.pdf

Realtime scheduling

The Linux provides two realtime scheduling policies : SCHED_RR and SCHED_FIFO.
The main concept is based on the priority levels, and transition between those.
The both policies are similar in terms that they need to define the rules about the fallowing use-cases:

- *Running* thread is preempted by the thread of a higher priority. After interrupted thread becomes ready again, how it will appear in the priority list (SCHED_FIFO: at the head)
- Thread is *blocked*, waiting on the synchronization primitive (like condition variable) to be signaled.

After being unblocked, and go to *ready* state: how it will appear in the belonging priority level list (SCHED_FIFO: at the tail).
(One well-known example could be the *priority inversion*, in case of the mutex - as synchronization primitive between two threads: one of low and other of high priority, causing the high priority thread starvation in case that the mutex is acquired by the low priority thread being preempted by the third: middle priority thread)

- Priority of a thread is changed explicitly. Depends on the weather priority is
  - increased (SCHED_FIFO: goes at the tail of the new priority list)
  - decreased (SCHED_FIFO: goes at the head of the new priority list)

The levels are [1-99], where 1 is the lowest, and 99 is the highest priority level.
To check in terminal, the realtime threads maximal priority range

```
# ulimit -r
```

When new thread is created and launched, it **inherits** by default the attributes of the parent thread, including the scheduling policy.
With POSIX-like APIs, there is a way to set scheduling policy **explicitly**, with `pthread_attr_setinheritsched:`

```
namespace pthread
{

    typedef void* (*thread_f)(void*);

    // Using C POSIX APIs

    int createThreadWithPrio( pthread_t* handle, thread_f func, void* context, int policy, int priority )
    {
        int err = 0;

        try
        {
                pthread_attr_t attr;
                err = pthread_attr_init(&attr);
                if (err)[[unlikely]]
                {
                    throw std::runtime_error("Failed: 'pthread_attr_init()'");
                }

                // Set the realtime thread schedule policy

                err = pthread_attr_setschedpolicy(&attr, policy);
                if (err)[[unlikely]]
                {
                    throw std::runtime_error("Failed: 'pthread_attr_setschedpolicy()'");
                }

               // Set the priority

                struct sched_param param;
                param.sched_priority = priority;
                err = pthread_attr_setschedparam(&attr, &param);
                if (err)[[unlikely]]
                {
                    throw std::runtime_error("Failed: 'pthread_attr_setschedparam()'");
                }

                // For this to take into account, the explicit scheduling needs to be specified.
                // Otherwise, the attributes will not be applied - the thread will inherit
                // the process/parent thread scheduling policy.
                // @note: This fails, if the user is unprivileged one (without CAP_SYS_NICE flag)
                err = pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
                if (err)[[unlikely]]
                {
                    throw std::runtime_error("Failed: 'pthread_attr_setinheritsched()'");
                }

                // Create thread with a given attributes
                err = pthread_create(handle, &attr, func, context);
                if (err)[[unlikely]]
                {
                    throw std::runtime_error("Failed: 'pthread_create()'");
                }

                check(handle, &attr);

                pthread_attr_destroy(&attr);

        }
        catch(const std::runtime_error& e)
        {
                details::log("<Thread> Exception: ", e.what());
                details::log("Error: ", err, ", ", strerror(err));
        }
```

```
        return err;
    }
}
```

The fact is, we can't set the attributes with *std::thread*.
The way to overcome this limitation, is to launch the POSIX thread as a **parent** one: setting there scheduling/priority explicitly, as an inheritable context in which thread function will be actually called

```
template <typename Func, typename... Args>
[[maybe_unused]] inline ThreadWrapper::ThreadWrapper(
        utils::ThreadWrapper::schedule_policy_t policy,
        utils::ThreadWrapper::priority_t priority,
        Func&& func,
        Args&&... args)
        : std::thread()
{
        auto threadFunc = std::bind(func, std::forward<Args>(args)..., std::placeholders::_1);
        // This may throw!
        std::ignore = pthread::createThreadWithPrio(
                native_handle(),
                threadFunc,
                nullptr,
                std::underlying_type_t<schedule_policy_t>(policy),
                priority);
}
```

 # Code to play with
https://godbolt.org/z/5b3GbdzTT

CPU Affinity

Today's embedded systems are mostly multiprocessor (multi-core) architectures.
In order to prevent performance impact caused by migrating threads from one CPU to another (invalidation of cache data), one can specified the affinity to a particular CPU.
This is especially important for time-critical processes and threads (like audio/video processing).

To list the available CPUs on Linux
```
# cat /proc/cpuinfo
```

To set the affinity programmatically, there are Linux nonstandard system calls
- *sched_setaffinity*()
- *sched_getaffinity*()

along with the macros CPU_ZERO, CPU_SET, CPU_CLR, CPU_ISSET

[The Linux Programming Interface (man7.org)](https://man7.org)

 # Code snippet

```
inline bool setAffinity(std::optional<int> core)
{
    const auto num_cpus = std::thread::hardware_concurrency();

    if (core)  // value set
    {
        if (*core < 0 || *core > static_cast<int>(num_cpus)) return false;
        return 0 == setAffinity(native_handle(), *core);
    }
     // core not specified: set the current CPU as designated one: prevents thread migration
    const auto core_id = sched_getcpu();
    return 0 == setAffinity(native_handle(), core_id);
}

inline int setAffinity(handle_t handle, int core)
{
    cpu_set_t cpuset;

    CPU_ZERO(&cpuset);
    CPU_SET(core, &cpuset);
    #ifdef __ANDROID__
        const auto tid = pthread_gettid_np(handle); // current thread id - std::this_thread::get_id()
        return sched_setaffinity(tid, sizeof(cpu_set_t), &cpuset);
    #else
        return pthread_setaffinity_np(handle, sizeof(cpu_set_t), &cpuset);
    #endif
}
```

In Linux, one can set/get (for verification) the affinity in terminal, with *taskset* command

To get affinity mask assigned to process
```
# taskset -p <pid>
```

To set affinity mask for a process
```
# taskset -p mask <pid>
```

[taskset(1) - Linux manual page (man7.org)](https://man7.org)

Android

If you are developing on Android, and having the **native (C/C++) code within .apk**, and you want to set the thread scheduling/priority for realtime threads (SCHED_RR/SCHED_FIFO), this would be miserably fail, since you most likely run as ***unprivileged*** process.
Even if the *SELinux* is turned into *permission* mode during development phase, the fact is that kernel in implementation of the POSIX-like system APIs that try to alter the thread scheduling/priority values (like *pthread_setschedparam*), check whether the **CAP_SYS_NICE** capability is enabled for the calling user.
https://man7.org/linux/man-pages/man7/capabilities.7.html

The ugly way to overcome this limitation, is to attach the native thread to the Java one: to set the priority (niceness) from the context of the Java thread, by calling ***android.os.Process.setThreadPriority***

```
namespace jni
{
    /**
     * Setting the thread priority (niceness) within Java thread context,
     * by calling the Process.setThreadPriority
     *
     * @param env        Pointer to the JNI function table
     * @param priority   Priority (niceness) to set
     * @return Indication of the operation outcome, TRUE on success.
     */
    static bool setThreadPriority(JNIEnv* env, priority_t priority)
    {
        try
        {
            jclass cls = env->FindClass("android/os/Process");
            if (nullptr == cls) throw std::runtime_error("<Thread> Invalid cls name.");
            jmethodID id = env->GetStaticMethodID(cls, "setThreadPriority", "(I)V");
            if (nullptr == id) throw std::runtime_error("<Thread> Invalid method id.");

            env->CallStaticVoidMethod(
                   cls,
                   id,
                   static_cast<jint>(priority));
               }
        catch (const std::runtime_error& e)
        {
            if (env->ExceptionCheck())
            {
                env->ExceptionDescribe();
                env->ExceptionClear();
            }

            ERROR_FMT("<Thread> Error: %s", e.what());

            return false;
        }

        return true;
    }
}   // namespace jni
```

More on JNI utility classes
https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/JNI%20Interface.pdf

This way, out *ThreadWrapper* gets additional constructor overloaded version

```
template <typename Func, typename... Args>
[[maybe_unused]] inline ThreadWrapper::ThreadWrapper(
        JavaVM* jvm,
        priority_t priority,
        std::string name,
        Func&& func,
        Args&&... args)
        : std::thread(
                [=, func_ = std::forward<Func>(func), name_ = std::move(name)](Args&&... args)
```

```
                {
                        jni::JNIThreadAnchor threadAnchor{jvm};
                        if (!threadAnchor) throw std::runtime_error("<Thread> Failed to attach native thread!");

                        // Set priority (niceness): at Java side, otherwise EPERM will be returned
                        if (!jni::setThreadPriority(threadAnchor.get(), priority))
                                throw std::runtime_error("<Thread> Failed to set priority!");

                        // Set name: at native side
                        std::ignore = setName(name_);
                        // Native thread function
                        std::invoke(func_, std::forward<Args>(args)...);
                },
                std::forward<Args>(args)...)
{}
```

If you check on the terminal, you'll see that this actually works

```
#!/bin/bash

pid=$(pidof $1)

while true
do
    ps -ATO SCH,PRI,NI -p "$pid"
    sleep 0.5
done
```

```
USER         PID   TID  PPID      VSZ    RSS WCHAN            ADDR S SCH PRI  NI CMD
u10_a120    4865  4865   481 15391176 190356 ep_poll             0 S   0  27  -8 [redacted]
u10_a120    4865  4873   481 15391176 190356 do_sigtim+          0 S   0  39 -20 Signal Catcher
u10_a120    4865  4874   481 15391176 190356 pipe_read           0 S   0  39 -20 perfetto_hprof_
u10_a120    4865  4875   481 15391176 190356 poll_sche+          0 S   0  39 -20 ADB-JDWP Connec
u10_a120    4865  4876   481 15391176 190356 futex_wai+          0 S   0  10   9 Jit thread pool
u10_a120    4865  4877   481 15391176 190356 futex_wai+          0 S   0  15   4 HeapTaskDaemon
u10_a120    4865  4878   481 15391176 190356 futex_wai+          0 S   0  15   4 ReferenceQueueD
u10_a120    4865  4879   481 15391176 190356 futex_wai+          0 S   0  15   4 FinalizerDaemon
u10_a120    4865  4880   481 15391176 190356 futex_wai+          0 S   0  15   4 FinalizerWatchd
u10_a120    4865  4881   481 15391176 190356 binder_th+          0 S   0  19   0 Binder:4865_1
u10_a120    4865  4882   481 15391176 190356 binder_th+          0 S   0  19   0 Binder:4865_2
u10_a120    4865  4885   481 15391176 190356 futex_wai+          0 S   0  10   9 Profile Saver
u10_a120    4865  4886   481 15391176 190356 ep_poll             0 S   0  29 -10 [redacted]
u10_a120    4865  4889   481 15391176 190356 futex_wai+          0 S   0  19   0 TraceClient
u10_a120    4865  4890   481 15391176 190356 futex_wai+          0 S   0  19   0 traceWD
```

@*note* The other way to check it is with *top* command:
#top -H -p $(pidof <process>)
but this doesn't show the scheduling policy

This reveals the bitter true: this workaround works only partially, and it's related with limitation that is imposed by the underlying system (Android) itself.
Unfortunately, in Android only supported scheduling at app level is **SCHED_OTHER (0)**, as default scheduling policy.
This means, you can only set the *niceness*, within the predefined range - as explained in the previous section.

→ **Digression**

Actually, Android uses it along with the CGROUP (Control Group) to enforce the two main priority categories:

- *background*
- *foreground*

threads.
There is ActivityManager that monitors the application status, and in case that app loses the focus, move the all tasks from one: foreground to other: background group, and vice versa.

For those who write the **vendor specific service**, that also contain native code, one way to overcome the privilege issue is to write the matching ***init.rc*** file, specifying the required capabilities
https://source.android.com/docs/core/permissions/ambient