User Managed Concurrency Groups (UMCG)

User Managed Concurrency Groups (UMCG) is an M:N threading subsystem/toolkit that lets user space application developers implement in-process user space schedulers.

Why? Heterogeneous in-process workloads

Linux kernel's CFS scheduler is designed for the "common" use case, with efficiency/throughput in mind. Work isolation and workloads of different "urgency" are addressed by tools such as cgroups, CPU affinity, priorities, etc., which are difficult or impossible to efficiently use in-process.

For example, a single DBMS process may receive tens of thousands requests per second; some of these requests may have strong response latency requirements as they serve live user requests (e.g. login authentication); some of these requests may not care much about latency but must be served within a certain time period (e.g. an hourly aggregate usage report); some of these requests are to be served only on a best-effort basis and can be NACKed under high load (e.g. an exploratory research/hypothesis testing workload).

Beyond different work item latency/throughput requirements as outlined above, the DBMS may need to provide certain guarantees to different users; for example, user A may "reserve" 1 CPU for their high-priority/low latency requests, 2 CPUs for mid-level throughput workloads, and be allowed to send as many best-effort requests as possible, which may or may not be served, depending on the DBMS load. Besides, the best-effort work, started when the load was low, may need to be delayed if suddenly a large amount of higher-priority work arrives. With hundreds or thousands of users like this, it is very difficult to guarantee the application's responsiveness using standard Linux tools while maintaining high CPU utilization.

Gaming is another use case: some in-process work must be completed before a certain deadline dictated by frame rendering schedule, while other work items can be delayed; some work may need to be cancelled/discarded because the deadline has passed; etc.

User Managed Concurrency Groups is an M:N threading toolkit that allows constructing user space schedulers designed to efficiently manage heterogeneous in-process workloads described above

while maintaining high CPU utilization (95%+).

Requirements

One relatively established way to design high-efficiency, low-latency systems is to split all work into small on-cpu work items, with asynchronous I/O and continuations, all executed on a thread pool with the number of threads not exceeding the number of available CPUs. Although this approach works, it is quite difficult to develop and maintain such a system, as, for example, small continuations are difficult to piece together when debugging. Besides, such asynchronous callback-based systems tend to be somewhat cache-inefficient, as continuations can get scheduled on any CPU regardless of cache locality.

M:N threading and cooperative user space scheduling enables controlled CPU usage (minimal OS preemption), synchronous coding style, and better cache locality.

Specifically:

- a variable/fluctuating number M of "application" threads should be "scheduled over" a relatively fixed number N of "kernel" threads, where N is less than or equal to the number of CPUs available;
- only those application threads that are attached to kernel threads are scheduled "on CPU";
- application threads should be able to cooperatively yield to each other;
- when an application thread blocks in kernel (e.g. in I/O), this becomes a scheduling event ("block") that the userspace scheduler should be able to efficiently detect, and reassign a waiting application thread to the freeded "kernel" thread;
- when a blocked application thread wakes (e.g. its I/O operation completes), this even ("wake") should also be detectable by the userspace scheduler, which should be able to either quickly dispatch the newly woken thread to an idle "kernel" thread or, if all "kernel" threads are busy, put it in the waiting queue;
- in addition to the above, it would be extremely useful for a separate in-process "watchdog" facility to be able to monitor the state of each of the M+N threads, and to intervene in case of runaway workloads (interrupt/preempt).

The building blocks

Main Objects

Based on the requirements above, UMCG exposes the following "objects":

- server tasks/threads: these are the N "kernel" threads from the requirements section above;
- worker tasks/threads: these are the M application threads from the requirements section above;
- UMCG groups: all UMCG worker and server threads belong to a UMCG group; a process (a shared MM) can have multiple groups; workers and servers must belong to the same UMCG group to interact (note: multiple groups per process can be useful to e.g. partition scheduling per NUMA node).

Key operations/API

As described above, the framework/toolkit must be able to efficiently process block/wake events, run workers, and provide cooperative worker scheduling facilities. As such, there are five runtime operations (not including control/support facilities), two for servers (explicit scheduling), and three for workers (cooperative scheduling):

- (server) run_worker(): a server specifies which worker to run; the call blocks; when the worker the server is running blocks in the kernel, the call returns, telling the server that its worker has blocked;
- (server) poll_worker(): a server polls the kernel for workers whose blocking operations has completed; the call returns the worker who woke the earliest (or blocks until there is such a worker);
- (worker) wait(): a worker can cooperatively "yield"; its attached server's run_worker() call returns;
- (worker) wake(): any task/thread can "wake" a worker that has yielded;
- (worker) swap(): a running worker can cooperatively "swap" its server with another worker.

Detailed state transitions are described below.

umcg_context_switch

This subsection explains some kernel-internal details.

It is important to emphasize that for a userspace scheduling framework to be of use, it is essential that common scenarios such as

- worker W1 blocks, wakes its serving server S which then runs worker W2
- worker W1 swaps into worker W2
- worker W unblocks, wakes a polling server S which then runs it

be as fast and efficient as possible. If these operations mean simply "wake remote task T and go to sleep", with T scheduled on a remote (idle) CPU, the overall performance of the system will be poor due to wakeup delays and cache locality issues (e.g. a server on CPU_A processes a worker blocked on CPU_B).

umcg_context_switch() is basically "wake remote task T on the current CPU and context switch into it"; it is a kernel-internal function that most operations outlined above use; it is exposed to the userspace indirectly via run_worker() (= context switch from the current server to the worker) and swap() (= context switch between two workers).

Initially, umcg_context_switch() is implemented by adding WF_CURRENT_CPU flag that is passed to ttwu; this change of the "wake remotely and go to sleep" operation to "wake on the current CPU and go to sleep" reduces the overall latency of swap about 8x on average (6-10 usec to less than 1 usec).

Another use case: fast IPC

Fast, synchronous on-CPU context switching can also be used for fast IPC (cross-process). For example, a typical security wrapper intercepts syscalls of an untrusted process, consults with external (out-of-process) "syscall firewall", and then delivers the allow/deny decision back (or the remote process actually proxies the syscall execution on behalf of the monitored process). This roundtrip is usually relatively slow, consuming at least 5-10 usec, as it involves waking a task on a remote CPU. A fast on-CPU context switch not only helps with the wakeup latency, but also has beneficial cache locality properties.

UMCG addresses this use case by providing another type of UMCG task: a "core" task. A core UMCG task can be thought of as a UMCG worker that does not belong to a UMCG group and that runs without a UMCG server attached to it, but that has access to the same UMCG worker operations, namely wait/wake/swap.

Userspace API

This section outlines the key components of UMCG API.

UMCG task states and state transitions

At a high level, UMCG is a task/thread managing/scheduling framework. The following task states are defined in uapi/linux/umcg.h (extra state flags are omitted here):

```
#define UMCG_TASK_NONE                0
/* UMCG server states. */
#define UMCG_TASK_POLLING             1
#define UMCG_TASK_SERVING             2
#define UMCG_TASK_PROCESSING          3
/* UMCG worker states. */
#define UMCG_TASK_RUNNABLE            4
#define UMCG_TASK_RUNNING             5
#define UMCG_TASK_BLOCKED             6
#define UMCG_TASK_UNBLOCKED           7
```

Server states and state transitions are easy:

- UMCG_TASK_POLLING: the server task is blocked in umcg_poll_worker(), waiting for an UNBLOCKED worker;
- UMCG_TASK_SERVING: the server task is blocked in umcg_run_worker(), serving a RUNNING worker;
- UMCG_TASK_PROCESSING: the server task is running in the userspace, presumably processing worker events.

Worker states and state transitions are more complicated:

- UMCG_TASK_RUNNING: the worker task is runnable/schedulable from the OS scheduler point of view (and is most likely running on a CPU);
- UMCG_TASK_RUNNABLE: this is a special worker state indicating that the worker is not runnable/schedulable by the OS scheduler, but can be scheduled by the user space scheduler; it is a sort of "voluntarily blocked" state;
- UMCG_TASK_BLOCKED: a previously RUNNING worker involuntarily blocked in the kernel, e.g. in a synchronous I/O operation; not runnable/schedulable by the OS scheduler;
- UMCG_TASK_UNBLOCKED: the blocking operation of a BLOCKED worker has completed, but the user space has not yet been notified of the event; not runnable/schedulable by the OS scheduler.

State transitions:

```
RUNNABLE => RUNNING : umcg_run_worker()
RUNNABLE => RUNNING : umcg_swap() (RUNNABLE next becomes RUNNING)

RUNNING  => RUNNABLE: umcg_swap() (RUNNING current becomes RUNNABLE)
RUNNING  => RUNNABLE: umcg_wait() (RUNNING current becomes RUNNABLE)
```

```
RUNNING   => BLOCKED  : the worker blocks in the kernel
BLOCKED   => UNBLOCKED: the worker's blocking operation has completed
UNBLOCKED => RUNNABLE : umcg_poll_worker()

RUNNABLE  => UNBLOCKED: umcg_wake()
```

Block/wake events are delivered to server tasks in userspace by waking them from their blocking operations:

- umcg_run_worker() returns (wakes the server) when the worker either blocks voluntarily via umcg_wait(), RUNNABLE, or involuntarily, BLOCKED;
- umcg_poll_worker() returns (wakes the server) when a new UNBLOCKED worker becomes available; if more then one UNBLOCKED worker is present, they are queued and umcg_poll_worker() returns immediately, with the longest waiting worker.

Several optimized state transitions will be possible in later versions of UMCG API. For example, it will be possible to call umcg_run_worker() on a BLOCKED worker so that BLOCKED => UNBLOCKED => RUNNABLE => RUNNING chain of states is "short cut" and happens behind the scenes. Or umcg_poll_worker() will be able to immediately run the polled worker, expediting UNBLOCKED => RUNNABLE => RUNNING.

The state transitions described above become somewhat more complicated in the presence of interrupts/signals, and the exact behavior in the presence of signals/interrupts is still work-in-progress.

UMCG core tasks have only two states: RUNNABLE and RUNNING; block/unblock detection logic is not applicable: if a UMCG core task blocks on I/O, it is still considered RUNNING. In a way, UMCG core tasks participate in cooperative scheduling, in that they can yield to each other via umcg_wait() and umcg_swap() and wake RUNNABLE tasks via umcg_wake().

Please note that while UMCG server/worker tasks must belong to the same UMCG group, and thus the same user process, UMCG core tasks can interact with each other across process boundaries.

UMCG API

- Runtime/scheduling:
  - Server API (explicit user space scheduling):
    - umcg_run_worker
    - umcg_poll_worker
  - Core/Worker API (cooperative scheduling):
    - umcg_wait
    - umcg_wake
    - umcg_swap
- Management:
  - umcg_create_group
  - umcg_destroy_group
  - umcg_register_task
  - umcg_unregister_task

API levels: porcelain and plumbing

Similarly to Git Porcelain and Plumbing API, UMCG exposes two API "surfaces": a higher-level "porcelain" API via libumcg, and a lower-level "plumbing" API via syscalls.

This design helps keep UMCG syscalls relatively lightweight, while hiding many implementation details from end users inside libumcg. It can be useful to think of libumcg as the only true UMCG API for end-users, while UMCG syscalls are a low-level toolkit that enables the higher-level API.

Porcelain API: libumcg

Located in $KDIR/tools/lib/umcg, libumcg exposes the following key API functions in libumcg.h (some non-essential helper functions are omitted here):

```
typedef intptr_t umcg_t;   /* UMCG group ID. */
typedef intptr_t umcg_tid; /* UMCG thread ID. */

#define UMCG_NONE      (0)

/**
 * umcg_get_utid - return the UMCG ID of the current thread.
 *
 * The function always succeeds, and the returned ID is guaranteed to be
 * stable over the life of the thread (and multiple
 * umcg_register/umcg_unregister calls).
 *
 * The ID is NOT guaranteed to be unique over the life of the process.
 */
umcg_tid umcg_get_utid(void);

/**
 * umcg_register_core_task - register the current thread as a UMCG core task
 *
 * Return:
 * UMCG_NONE     - an error occurred. Check errno.
 * != UMCG_NONE  - the ID of the thread to be used with UMCG API (guaranteed
 *                 to match the value returned by umcg_get_utid).
 */
umcg_tid umcg_register_core_task(intptr_t tag);

/**
 * umcg_register_worker - register the current thread as a UMCG worker
 * @group_id:      The ID of the UMCG group the thread should join.
 *
 * Return:
 * UMCG_NONE     - an error occurred. Check errno.
 * != UMCG_NONE  - the ID of the thread to be used with UMCG API (guaranteed
 *                 to match the value returned by umcg_get_utid).
 */
umcg_tid umcg_register_worker(umcg_t group_id, intptr_t tag);

/**
 * umcg_register_server - register the current thread as a UMCG server
 * @group_id:      The ID of the UMCG group the thread should join.
 *
 * Return:
 * UMCG_NONE     - an error occurred. Check errno.
 * != UMCG_NONE  - the ID of the thread to be used with UMCG API (guaranteed
 *                 to match the value returned by umcg_get_utid).
 */
umcg_tid umcg_register_server(umcg_t group_id, intptr_t tag);

/**
 * umcg_unregister_task - unregister the current thread
```

```
 *
 * Return:
 * 0               - OK
 * -1              - the current thread is not a UMCG thread
 */
int umcg_unregister_task(void);

/**
 * umcg_wait - block the current thread
 * @timeout:   absolute timeout (not supported at the moment)
 *
 * Blocks the current thread, which must have been registered via umcg_register,
 * until it is woken via umcg_wake or swapped into via umcg_swap. If the current
 * thread has a wakeup queued (see umcg_wake), returns zero immediately,
 * consuming the wakeup.
 *
 * Return:
 * 0         - OK, the thread was waken;
 * -1        - did not wake normally;
 *               errno:
 *                 EINTR: interrupted
 *                 EINVAL: some other error occurred
 */
int umcg_wait(const struct timespec *timeout);

/**
 * umcg_wake - wake @next
 * @next:      ID of the thread to wake (IDs are returned by umcg_register).
 *
 * If @next is blocked via umcg_wait, or umcg_swap, wake it. If @next is
 * running, queue the wakeup, so that a future block of @next will consume
 * the wakeup but will not block.
 *
 * umcg_wake is non-blocking, but may retry a few times to make sure @next
 * has indeed woken.
 *
 * umcg_wake can queue at most one wakeup; if @next has a wakeup queued,
 * an error is returned.
 *
 * Return:
 * 0         - OK, @next has woken, or a wakeup has been queued;
 * -1        - an error occurred.
 */
int umcg_wake(umcg_tid next);

/**
 * umcg_swap - wake @next, put the current thread to sleep
 * @next:      ID of the thread to wake
 * @timeout:   absolute timeout (not supported at the moment)
 *
 * umcg_swap is semantically equivalent to
 *
 *     int ret = umcg_wake(next);
 *     if (ret)
 *             return ret;
 *     return umcg_wait(timeout);
 *
 * but may do a synchronous context switch into @next on the current CPU.
 */
int umcg_swap(umcg_tid next, const struct timespec *timeout);
```

```
/**
 * umcg_create_group - create a UMCG group
 * @flags:             Reserved.
 *
 * UMCG groups have worker and server threads.
 *
 * Worker threads are either RUNNABLE/RUNNING "on behalf" of server threads
 * (see umcg_run_worker), or are BLOCKED/UNBLOCKED. A worker thread can be
 * running only if it is attached to a server thread (interrupts can
 * complicate the matter - TBD).
 *
 * Server threads are either blocked while running worker threads or are
 * blocked waiting for available (=UNBLOCKED) workers. A server thread
 * can "run" only one worker thread.
 *
 * Return:
 * UMCG_NONE     - an error occurred. Check errno.
 * != UMCG_NONE  - the ID of the group, to be used in e.g. umcg_register.
 */
umcg_t umcg_create_group(uint32_t flags);

/**
 * umcg_destroy_group - destroy a UMCG group
 * @umcg:               ID of the group to destroy
 *
 * The group must be empty (no server or worker threads).
 *
 * Return:
 * 0            - Ok
 * -1           - an error occurred. Check errno.
 *                errno == EAGAIN: the group has server or worker threads
 */
int umcg_destroy_group(umcg_t umcg);

/**
 * umcg_poll_worker - wait for the first available UNBLOCKED worker
 *
 * The current thread must be a UMCG server. If there is a list/queue of
 * waiting UNBLOCKED workers in the server's group, umcg_poll_worker
 * picks the longest waiting one; if there are no UNBLOCKED workers, the
 * current thread sleeps in the polling queue.
 *
 * Return:
 * UMCG_NONE         - an error occurred; check errno;
 * != UMCG_NONE      - a RUNNABLE worker.
 */
umcg_tid umcg_poll_worker(void);

/**
 * umcg_run_worker - run @worker as a UMCG server
 * @worker:          the ID of a RUNNABLE worker to run
 *
 * The current thread must be a UMCG "server".
 *
 * Return:
 * UMCG_NONE    - if errno == 0, the last worker the server was running
 *                unregistered itself; if errno != 0, an error occurred
 * != UMCG_NONE - the ID of the last worker the server was running before
 *                the worker was blocked or preempted.
 */
umcg_tid umcg_run_worker(umcg_tid worker);
```

```
/**
 * umcg_get_task-state - return the current UMCG state of @task
 * @task:                the ID of a UMCG task
 *
 * Return: one of UMCG_TASK_*** values defined in uapi/linux/umcg.h
 */
uint32_t umcg_get_task_state(umcg_tid task);
```

Plumbing API: sys_umcg_*** syscalls

The following new Linux syscalls are proposed:

```
sys_umcg_api_version
sys_umcg_register_task
sys_umcg_unregister_task
sys_umcg_wait
sys_umcg_wake
sys_umcg_swap
sys_umcg_create_group
sys_umcg_destroy_group
sys_umcg_poll_worker
sys_umcg_run_worker
sys_umcg_preempt_worker
```

They closely resemble umcg_*** functions from libumcg, but are more lightweight. For example, some of the state transitions described above are technically handled in the user space (libumcg), with the syscalls doing the absolute minimum. The exact boundary of what is done in the kernel and what is delegated to the userspace is still work-in-progress. End users are supposed to use libumcg rather than call sys_umcg_*** directly.

sys_umcg_preempt_worker() is a placeholder for a new syscall to be added in the future.

A brief historical note

In 2012-2013 Paul Turner and his team at Google developed SwitchTo and SwitchTo Groups Linux kernel extensions, on top of which a C++ user space scheduling framework called "Google Fibers" is built; it is used widely and successfully at Google.

UMCG core API (wait/wake/swap) is based on SwitchTo API, while the overall UMCG API resembles SwitchTo Groups API.