UMCG Userspace API

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).

UMCG kernel API

Based on the requrements above, UMCG *kernel* API is build around the following ideas:

- *UMCG server*: a task/thread representing "kernel threads", or CPUs from the requirements above;
- *UMCG worker*: a task/thread representing "application threads", to be scheduled over servers;
- UMCG *task state*: (NONE), RUNNING, BLOCKED, IDLE: states a UMCG task (a server or a worker) can be in;
- UMCG task *state flag*: LOCKED, PREEMPTED: additional state flags that can be ORed with the task state to communicate additional information to the kernel;
- `struct umcg_task`: a per-task userspace set of data fields, usually residing in the TLS, that fully reflects the current task's UMCG state and controls the way the kernel manages the task;
- `sys_umcg_ctl()`: a syscall used to register the current task/thread as a server or a worker, or to unregister a UMCG task;
- `sys_umcg_wait()`: a syscall used to put the current task to sleep and/or wake another task, pontentially context-switching between the two tasks on-CPU synchronously.

Servers

When a task/thread is registered as a server, it is in RUNNING state and behaves like any other normal task/thread. In addition, servers can interact with other UMCG tasks via sys_umcg_wait():

- servers can voluntarily suspend their execution (wait), becoming IDLE;
- servers can wake other IDLE servers;
- servers can context-switch between each other.

Note that if a server blocks in the kernel *not* via sys_umcg_wait(), it still retains its RUNNING state.

Also note that servers can be used for fast on-CPU context switching across process boundaries; server-worker interactions assume they belong to the same mm.

See the next section on how servers interact with workers.

Workers

A worker cannot be RUNNING without having a server associated with it, so when a task is first registered as a worker, it enters the IDLE state.

- a worker becomes RUNNING when a server calls sys_umcg_wait to context-switch into it; the server goes IDLE, and the worker becomes RUNNING in its place;
- when a running worker blocks in the kernel, it becomes BLOCKED, its associated server becomes RUNNING and the server's sys_umcg_wait() call from the bullet above returns; this transition is sometimes called "block detection";
- when the syscall on which a BLOCKED worker completes, the worker becomes IDLE and is added to the list of idle workers; if there is an idle server waiting, the kernel wakes it; this transition is sometimes called "wake detection";
- running workers can voluntarily suspend their execution (wait), becoming IDLE; their associated servers are woken;
- a RUNNING worker can context-switch with an IDLE worker; the server of the switched-out worker is transferred to the switched-in worker;
- any UMCG task can "wake" an IDLE worker via sys_umcg_wait(); unless this is a server running the worker as described in the first bullet in this list, the worker remain IDLE but is added to the idle workers list; this "wake" operation exists for completeness, to make sure wait/wake/context-switch operations are available for all UMCG tasks;
- the userspace can preempt a RUNNING worker by marking it `RUNNING|PREEMPTED` and sending a signal to it; the userspace should have installed a NOP signal handler for the signal; the kernel will then transition the worker into `IDLE|PREEMPTED` state and wake its associated server.

UMCG task states

Important: all state transitions described below involve at least two steps: the change of the state field in `struct umcg_task`, for example `RUNNING` to `IDLE`, and the corresponding change in `struct task_struct` state, for example a transition between the task running on CPU and being descheduled and removed from the kernel runqueue. The key principle of UMCG API design is that the party initiating the state transition modifies the state variable.

For example, a task going `IDLE` first changes its state from `RUNNING` to `IDLE` in the userpace and then calls `sys_umcg_wait()`, which completes the transition.

Note on documentation: in `include/uapi/linux/umcg.h`, task states have the form `UMCG_TASK_RUNNING`, `UMCG_TASK_BLOCKED`, etc. In this document these are usually referred to simply `RUNNING` and `BLOCKED`, unless it creates ambiguity. Task state flags, e.g. `UMCG_TF_PREEMPTED`, are treated similarly.

UMCG task states reflect the view from the userspace, rather than from the kernel. There are three fundamental task states:

- `RUNNING`: indicates that the task is schedulable by the kernel; applies to both servers and workers;
- `IDLE`: indicates that the task is *not* schedulable by the kernel (see `umcg_idle_loop()` in `kernel/sched/umcg.c`); applies to both servers and workers;
- `BLOCKED`: indicates that the worker is blocked in the kernel; does not apply to servers.

In addition to the three states above, two state flags help with state transitions:

- `LOCKED`: the userspace is preparing the worker for a state transition and "locks" the worker until the worker is ready for the kernel to act on the state transition; used similarly to preempt_disable or irq_disable in the kernel; applies only to workers in `RUNNING` or `IDLE` state; `RUNNING|LOCKED` means "this worker is about to become `RUNNING`, while `IDLE|LOCKED` means "this worker is about to become `IDLE` or unregister;
- `PREEMPTED`: the userspace indicates it wants the worker to be preempted; there are no situations when both `LOCKED` and `PREEMPTED` flags are set at the same time.

struct umcg_task

From `include/uapi/linux/umcg.h`:

```
struct umcg_task {
      uint32_t        state;                  /* r/w */
      uint32_t        next_tid;               /* r   */
      uint64_t        idle_workers_ptr;       /* r/w */
      uint64_t        idle_server_tid_ptr;    /* r*  */
};
```

Each UMCG task is identified by `struct umcg_task`, which is provided to the kernel when the task is registered via `sys_umcg_ctl()`.

- `uint32_t state`: the current state of the task this struct identifies, as described in the previous section. Readable/writable by both the kernel and the userspace.
    - bits 0 - 7: task state (RUNNING, IDLE, BLOCKED);
    - bits 8 - 15: state flags (LOCKED, PREEMPTED);
    - bits 16 - 23: reserved; must be zeroes;
    - bits 24 - 31: for userspace use.
- `uint32_t next_tid`: contains the TID of the task to context-switch-into in `sys_umcg_wait()`; can be zero; writable by the userspace, readable by the kernel; if this is a RUNNING worker, this field contains the TID of the server that should be woken when this worker blocks; see `sys_umcg_wait()` for more details;
- `uint64_t idle_workers_ptr`: this field forms a single-linked list of idle workers: all RUNNING workers have this field set to point to the head of the list (a pointer variable in the

userspace).

When a worker's blocking operation in the kernel completes, the kernel changes the worker's state from `BLOCKED` to `IDLE` and adds the worker to the top of the list of idle workers using this logic:

```
/* kernel side */
u64 *head = (u64 *)(worker->idle_workers_ptr); /* get the head pointer */
u64 *first = (u64 *)*head; /* get the first element */

/* make the worker's ptr point to the first element */
worker->idle_workers_ptr = first;

/* make the head pointer point to this worker */
if (cmpxchg(head, &first, &worker->idle_workers_ptr))
    /* success */
else
    /* retry, with exponential back-off */
```

In the userspace the list is cleared atomically using this logic:

```
/* userspace side */
uint64_t *idle_workers = (uint64_t *)*head;

/* move the list from the global head to the local idle_workers */
if (cmpxchg(&head, &idle_workers, NULL))
    /* success: process idle_workers */
else
    /* retry */
```

The userspace re-points workers' idle_workers_ptr to the list head variable before the worker is allowed to become RUNNING again.

- `uint64_t idle_server_tid_ptr`: points to a pointer variable in the userspace that points to an idle server, i.e. a server in IDLE state waiting in sys_umcg_wait(); read-only; workers must have this field set; not used in servers.

  When a worker's blocking operation in the kernel completes, the kernel changes the worker's state from `BLOCKED` to `IDLE`, adds the worker to the list of idle workers, and checks whether `*idle_server_tid_ptr` is not zero. If not, the kernel tries to cmpxchg() it with zero; if cmpxchg() succeeds, the kernel will then wake the server. See State transitions below for more details.

sys_umcg_ctl()

`int sys_umcg_ctl(uint32_t flags, struct umcg_task *self)` is used to register or unregister the current task as a worker or server. Flags can be one of the following:

- `UMCG_CTL_REGISTER`: register a server;
- `UMCG_CTL_REGISTER | UMCG_CTL_WORKER`: register a worker;
- `UMCG_CTL_UNREGISTER`: unregister the current server or worker.

When registering a task, `self` must point to `struct umcg_task` describing this server or worker; the pointer must remain valid until the task is unregistered.

When registering a server, `self->state` must be `RUNNING`; all other fields in `self` must be zeroes.

When registering a worker, `self->state` must be `IDLE`; `self->idle_server_tid_ptr` and `self->idle_workers_ptr` must be valid pointers as described in struct umcg_task; `self->next_tid` must be zero.

When unregistering a task, `self` must be `NULL`.

sys_umcg_wait()

`int sys_umcg_wait(uint32_t flags, uint64_t abs_timeout)` operates on registered UMCG servers and workers: `struct umcg_task *self` provided to `sys_umcg_ctl()` when registering the current task is consulted in addition to `flags` and `abs_timeout` parameters.

The function can be used to perform one of the three operations:

- wait: if `self->next_tid` is zero, `sys_umcg_wait()` puts the current server or worker to sleep;
- wake: if `self->next_tid` is not zero, and `flags & UMCG_WAIT_WAKE_ONLY`, the task identified by `next_tid` is woken (must be in `IDLE` state);
- context switch: if `self->next_tid` is not zero, and `!(flags & UMCG_WAIT_WAKE_ONLY)`, the current task is put to sleep and the next task is woken, synchronously switching between the tasks on the current CPU on the fast path.

Flags can be zero or a combination of the following values:

- `UMCG_WAIT_WAKE_ONLY`: wake the next task, don't put the current task to sleep;
- `UMCG_WAIT_WF_CURRENT_CPU`: wake the next task on the curent CPU; this flag has an effect only if `UMCG_WAIT_WAKE_ONLY` is set: context switching is always attempted to happen on the curent CPU.

The section below provides more details on how servers and workers interact via `sys_umcg_wait()`, during worker block/wake events, and during worker preemption.

State transitions

As mentioned above, the key principle of UMCG state transitions is that **the party initiating the state transition modifies the state of affected tasks**.

Below, "`TASK:STATE`" indicates a task T, where T can be either W for worker or S for server, in state S, where S can be one of the three states, potentially ORed with a state flag. Each individual state transition is an atomic operation (cmpxchg) unless indicated otherwise. Also note that **the order of state transitions is important and is part of the contract between the userspace and the kernel. The kernel is free to kill the task (SIGSEGV) if the contract is broken.**

Some worker state transitions below include adding `LOCKED` flag to worker state. This is done to indicate to the kernel that the worker is transitioning state and should not participate in the block/wake detection routines, which can happen due to interrupts/pagefaults/signals.

`IDLE|LOCKED` means that a running worker is preparing to sleep, so interrupts should not lead to server wakeup; `RUNNING|LOCKED` means that an idle worker is going to be "scheduled to run", but may not yet have its server set up properly.

Key state transitions:

- server to worker context switch ("schedule a worker to run"): `S:RUNNING+W:IDLE => S:IDLE+W:RUNNING`:
  - in the userspace, in the context of the server S running:
    - `S:RUNNING => S:IDLE` (mark self as idle)
    - `W:IDLE => W:RUNNING|LOCKED` (mark the worker as running)
    - `W.next_tid := S.tid; S.next_tid := W.tid` (link the server with the worker)
    - `W:RUNNING|LOCKED => W:RUNNING` (unlock the worker)
    - `S: sys_umcg_wait()` (make the syscall)
  - the kernel context switches from the server to the worker; the server sleeps until it becomes `RUNNING` during one of the transitions below;
- worker to server context switch (worker "yields"): `S:IDLE+W:RUNNING => S:RUNNING+W:IDLE`:
  - in the userspace, in the context of the worker W running (note that a running worker has its `next_tid` set to point to its server):
    - `W:RUNNING => W:IDLE|LOCKED` (mark self as idle)
    - `S:IDLE => S:RUNNING` (mark the server as running)
    - `W: sys_umcg_wait()` (make the syscall)
  - the kernel removes the `LOCKED` flag from the worker's state and context switches from the worker to the server; the worker sleeps until it becomes `RUNNING`;
- worker to worker context switch: `W1:RUNNING+W2:IDLE => W1:IDLE+W2:RUNNING`:
  - in the userspace, in the context of W1 running:
    - `W2:IDLE => W2:RUNNING|LOCKED` (mark W2 as running)
    - `W1:RUNNING => W1:IDLE|LOCKED` (mark self as idle)
    - `W2.next_tid := W1.next_tid; S.next_tid := W2.next_tid` (transfer the server W1 => W2)
    - `W1:next_tid := W2.tid` (indicate that W1 should context-switch into W2)
    - `W2:RUNNING|LOCKED => W2:RUNNING` (unlock W2)
    - `W1: sys_umcg_wait()` (make the syscall)
  - same as above, the kernel removes the `LOCKED` flag from the W1's state and context switches to next_tid;
- worker wakeup: `W:IDLE => W:RUNNING`:
  - in the userspace, a server S can wake a worker W without "running" it:
    - `S:next_tid :=W.tid`
    - `W:next_tid := 0`
    - `W:IDLE => W:RUNNING`
    - `sys_umcg_wait(UMCG_WAIT_WAKE_ONLY)` (make the syscall)
  - the kernel will wake the worker W; as the worker does not have a server assigned, "wake detection" will happen, the worker will be immediately marked as `IDLE` and added to idle workers list; an idle server, if any, will be woken (see 'wake detection' below);
  - Note: if needed, it is possible for a worker to wake another worker: the waker marks itself "IDLE|LOCKED", points its next_tid to the wakee, makes the syscall, restores its server in next_tid, marks itself as `RUNNING`.
- block detection: worker blocks in the kernel: `S:IDLE+W:RUNNING => S:RUNNING+W:BLOCKED`:
  - when a worker blocks in the kernel in `RUNNING` state (not `LOCKED`), before descheduling the task from the CPU the kernel performs these operations:
    - `W:RUNNING => W:BLOCKED`
    - `S := W.next_tid`
    - `S:IDLE => S:RUNNING`
    - `try_to_wake_up(S)`

    - if any of the first three operations above fail, the worker is killed via `SIGSEGV`. Note that `ttwu(S)` is not required to succeed, as the server may still be transitioning to sleep in `sys_umcg_wait()`; before actually putting the server to sleep its UMCG state is checked and, if it is `RUNNING`, sys_umcg_wait() returns to the userspace;
    - if the worker has its `LOCKED` flag set, block detection does not trigger, as the worker is assumed to be in the userspace scheduling code.
- wake detection: worker wakes in the kernel: `W:BLOCKED => W:IDLE`:
    - all workers' returns to the userspace are intercepted:
        - `start:` (a label)
        - if `W:RUNNING & W.next_tid != 0`: let the worker exit to the userspace, as this is a `RUNNING` worker with a server;
        - `W:* => W:IDLE` (previously blocked or woken without servers workers are not allowed to return to the userspace);
        - the worker is appended to `W.idle_workers_ptr` idle workers list;
        - `S := *W.idle_server_tid_ptr; if (S != 0) S:IDLE => S.RUNNING; ttwu(S)`
        - `idle_loop(W)`: this is the same idle loop that `sys_umcg_wait()` uses: it breaks only when the worker becomes `RUNNING`; when the idle loop exits, it is assumed that the userspace has properly removed the worker from the idle workers list before marking it `RUNNING`;
        - `goto start;` (repeat from the beginning).
    - the logic above is a bit more complicated in the presence of `LOCKED` or `PREEMPTED` flags, but the main invariants stay the same:
        - only `RUNNING` workers with servers assigned are allowed to run in the userspace (unless `LOCKED`);
        - newly `IDLE` workers are added to the idle workers list; any user-initiated state change assumes the userspace properly removed the worker from the list;
        - as with wake detection, any "breach of contract" by the userspace will result in the task termination via `SIGSEGV`.
- worker preemption: `S:IDLE+W:RUNNING => S:RUNNING+W:IDLE|PREEMPTED`:
    - when the userspace wants to preempt a `RUNNING` worker, it changes it state, atomically, `RUNNING => RUNNING|PREEMPTED` and sends a signal to the worker via `tgkill()`; the signal handler, previously set up by the userspace, can be a NOP (note that only `RUNNING` workers can be preempted);
    - if the worker, at the moment the signal arrived, continued to be running on-CPU in the userspace, the "wake detection" code will be triggered that, in addition to what was described above, will check if the worker is in `RUNNING|PREEMPTED` state:
        - `W:RUNNING|PREEMPTED => W:IDLE|PREEMPTED`
        - `S := W.next_tid`
        - `S:IDLE => S:RUNNING`
        - `try_to_wakeup(S)`
    - if the signal arrives after the worker blocks in the kernel, the "block detection" happened as described above, with the following change:
        - `W:RUNNING|PREEMPTED => W:BLOCKED|PREEMPTED`
        - `S := W.next_tid`
        - `S:IDLE => S:RUNNING`
        - `try_to_wake_up(S)`
    - in any case, the worker's server is woken, with its attached worker (`S.next_tid`) either in `BLOCKED|PREEMPTED` or `IDLE|PREEMPTED` state.

Server-only use cases

Some workloads/applications may benefit from fast and synchronous on-CPU user-initiated context switches without the need for full userspace scheduling (block/wake detection). These applications can use "standalone" UMCG servers to wait/wake/context-switch, including across process boundaries.

These "worker-less" operations involve trivial `RUNNING` <==> `IDLE` state changes, not discussed here for brevity.