Devsummit – Concurrency Hacks

Taylor ‘Riastradh’ Campbell
campbell@mumble.net
riastradh@NetBSD.org

EuroBSDcon 2015
Stockholm, Sweden
October 2, 2015
Concurrent hacks

Works in progress — not even compile-tested.

- Lightweight task queues
- Pserialized reader/writer locks
- Reference counts
**Tasks background – Softints**

- `softint(9)`: defer processing from hardware interrupt handlers to lower-priority to remain responsive to hardware interrupts.

- `softint = softint_establish(&mydriver_softintr, arg);

  ...

  softint_schedule(softint);

  
  ```c
  static void
  mydriver_softintr(void *arg)
  {
    ...
  }
  ```
Tasks background – Softints

- Limited number of softints: use sparingly.
- `softint_schedule` can’t pass argument.
- Multi-CPU: `softint_schedule` schedules up to one softint per CPU at a time, executed in parallel.
- Cheap: zero interprocessor synchronization.
Tasks background – Workqueues

► workqueue(9): defer processing from higher-priority threads to lower-priority threads.

► struct mydriver_softc {
    ...
    struct workqueue *sc_wq;
    struct work sc_wk;
};

error = workqueue_create(&sc->sc_wq, "mydriver", &mydriver_work, arg, PRI_NONE, IPL_NET, WQ_MPSAFE);
    ...
    workqueue_enqueue(sc->sc_wq, &sc->sc_wk, NULL);

static void
mydriver_work(struct work *wk, void *arg)
    ...

Tasks background – Workqueues

- Caller must not reuse struct work until done, must do necessary bookkeeping to avoid this.
- Caller can pass arguments by embedding struct work:
  ```c
  struct mywork {
      struct work w;
      int extra;
  };
  ```
- Multi-threaded (WQ_PERCPU): each struct work can execute in parallel, one worker thread per CPU.
- Caller can’t wait for individual work — can only destroy workqueue and wait for all.
- workqueue_enqueue acquires per-CPU mutex, so requires interprocessor synchronization.
- One dedicated thread per workqueue (per CPU): wastes kernel address space for mostly unused workqueues.
Tasks background – USB tasks

- `usb_task(9)`: defer processing from USB interrupt handler to thread.
- `struct udriver_softc {
  ...
  struct usb_task sc_task;
};`

  `usb_init_task(&sc->sc_task, &udriver_task, sc, USB_TASKQ_MPSAFE);
  ...
  usb_add_task(sc->sc_udev, &sc->sc_task, USB_TASKQ_DRIVER);
  ...
  usb_rem_task(sc->sc_udev, &sc->sc_task);

  static void
  udriver_task(void *arg)
  ...

Tasks background – USB Tasks

- USB-specific.
- Per-device/per-host-controller task queues.
- **BUG:** No way for task to complete — need this for driver detach.
- `usb_add_task` and `usb_rem_task` acquire shared mutex, so require interprocessor synchronization.
Tasks background – Callouts

- `callout(9)`: defer processing until ticks have passed.
- `hz` granularity.
- Unlike others, supports synchronous cancellation — `callout_halt`.
- Complex triggering protocol: `callout_pending`, `callout_ack`, `callout_expired`, ...
- ...

...
Tasks – Unified proposal

- `task(9)`: defer processing from higher-priority contexts to lower-priority contexts.
- `struct mydriver_softc {
   ...
   struct task sc_task;
};`

  `task_init(task, &mydriver_task);
  task_schedule(task);`

  `static void
mydriver_task(struct task *task)
{
   ...
  }`
 Tasks – Unified proposal

- Easy to use.
- Synchronous cancellation.
- Slightly more synchronization overhead softint: `task_schedule` acquires mutex — but only for local CPU, not contended unless doing cancel.
- Slightly more memory overhead than workqueue: Caller can reschedule struct `task` without problem (no effect), unlike workqueue struct `work`.
- Delayed tasks with nanosecond-resolution API, simpler triggering protocol.
Tasks – Explicit task queues

- Common API for softint and thread priority levels.
- Default shared system task queues at each softint and thread priority level.
- Guaranteed concurrency if you make your own task queue: not held up by other system tasks.
- Per-CPU thread pool shared by different task queues — no threads wasted on mostly unused task queues.
Pserialized reader/writer locks

- Example: fstrans — recursive transactions to block file system operations if operator requests suspend, e.g. to take snapshot.
- fstrans_begin/fstrans_end are cheap if no suspend in progress: no interprocessor synchronization, using pserialize(9).
- Suspend is expensive: not just interprocessor synchronization, but cross-call to wait for all transactions to drain.
- (Fstrans also handles establishing copy-on-write hooks.)
Pserialized reader/writer locks

- rrwlock(9) generalizes (part of) fstrans(9):
  - l = rrwlock_create("foo");
  - struct rrw_reader r; rrwlock_reader_enter(l, &r);
  - ...
  - rrwlock_reader_exit(l, r);
  - struct rrw_writer w; rrwlock_writer_enter(l, &w);
  - ...
  - rrwlock_writer_exit(l, w);
  - rrwlock_destroy(l);
  - Also non-recursive variant, prwlock(9).
Reference counts

- refcount(9): simple reference counts.
- Many copies of simple reference-counting logic: e.g., struct kauth_cred, struct ifaddr.
- Nothing novel or exciting here: just a nice API.

```c
/* Create object. */
refcount_init(&obj->refcount);

/* Acquire reference. */
refcount_inc(&obj->refcount);

/* Assert held. */
KASSERT(refcount_referenced_p(&obj->refcount));
```
Reference counts

* Release reference and free if last one. */
if (refcount_dec_local(&obj->refcount)) {
    kmem_free(obj, sizeof(*obj));
}

/* Release reference and notify waiters. */
refcount_dec_signal(&obj->refcount, &obj->lock);

/* Release reference and wait for other users. */
refcount_dec_drain(&obj->refcount, &obj->lock);
kmem_free(obj, sizeof(*obj));
Problem: cache, don’t free, on last reference

▶ Want to cache vnodes in memory to avoid reparsing disk for frequently referenced files.
▶ On last reference, put vnodes on queue to be freed when memory is tight.
▶ Need to synchronize between acquiring cached vnode and freeing cached vnode.
▶ Need to coordinate with per-file-system ‘delete file on last reference’ logic.
▶ refcount(9) API doesn’t help with this.