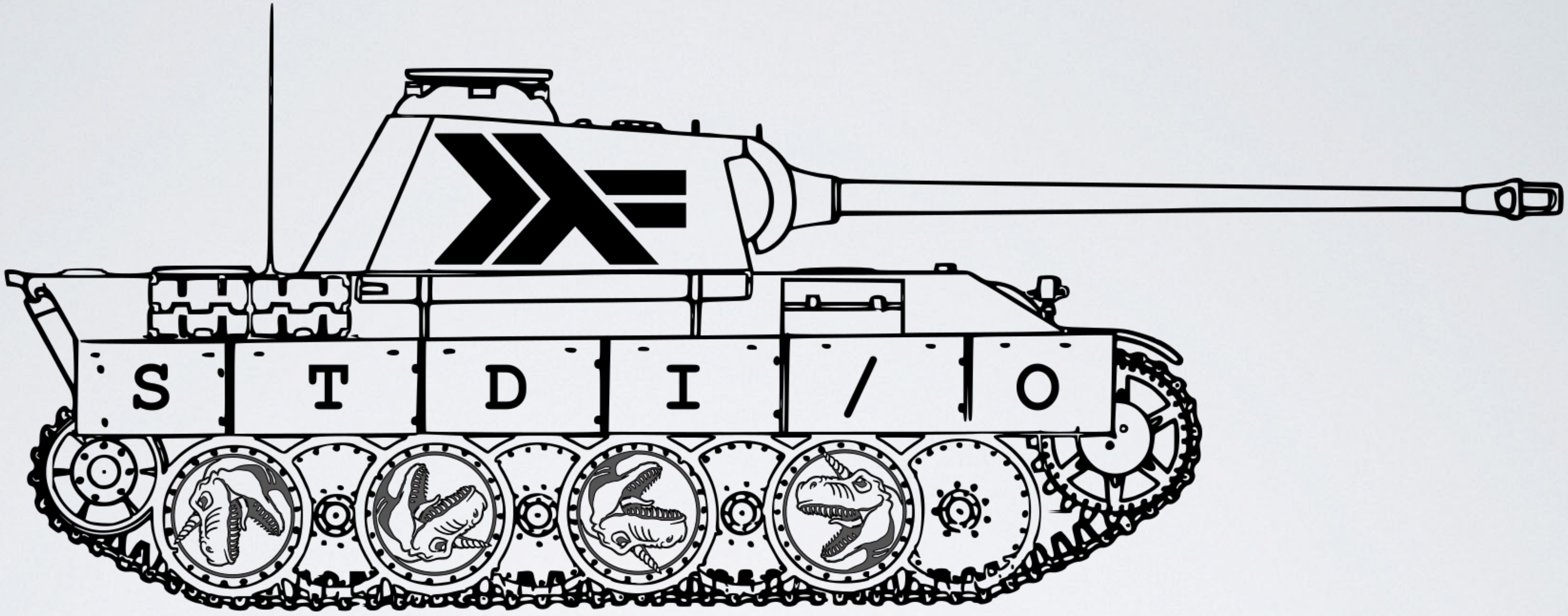

Libuv based I/O Manager

-- 韩冬(github.com/winterland1989)

The I/O schedule problem

- Modern OS manage work in unit of process/thread, and running threads often do I/O.
- I/O operation may block, e.g. stop one thread from running for an uncertain time.
- We want to process I/O concurrently as much as possible, e.g. threads doing I/O shouldn't stop others from doing their work.

The simplest I/O multiplexer

- Start new threads to do concurrent I/O.
- Use blocking system calls, let the kernel do the rest, e.g. auto put thread blocking on I/O to background, wake up when I/O finishes.
- Easy to write programs, perform well under small concurrent numbers.
- OS thread is rather expensive when concurrent number goes up.

Enter Event-driven

```
// introduced in 4.2BSD Unix, released in August 1983.
int select(int nfds, fd_set *readfds, fd_set *writefds
          , fd_set *exceptfds, struct timeval *timeout);
void FD_CLR(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
int  FD_ISSET(int fd, fd_set *set);

// introduced in SVR3 Unix, released 1986.
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
struct pollfd {
    int   fd;          /* file descriptor */
    short events;      /* requested events */
    short revents;     /* returned events */
};
```

Better event(epoll, kqueue…)

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd
             , struct epoll_event *event);
// op: EPOLL_CTL_ADD , EPOLL_CTL_MOD , EPOLL_CTL_DEL

struct epoll_event {
    uint32_t     events;      /* Epoll events */
    epoll_data_t data;        /* User data variable */
};

int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
```

Overlapped I/O

```
int WSARecv(
  _In_    SOCKET                s,
  _Inout_ LPWSABUF              lpBuffers,
  _In_    DWORD                 dwBufferCount,
  _Out_   LPDWORD               lpNumberOfBytesRecvd,
  _Inout_ LPDWORD               lpFlags,
  _In_    LPWSAOVERLAPPED       lpOverlapped,
  _In_    LPWSAOVERLAPPED_COMPLETION_ROUTINE
                                lpCompletionRoutine
);
DWORD WSAWaitForMultipleEvents(
  _In_       DWORD    cEvents,
  _In_ const WSAEVENT *lphEvents,
  _In_       BOOL     fWaitAll,
  _In_       DWORD    dwTimeout,
  _In_       BOOL     fAlertable
);
```

IOCP + Overlapped I/O

```
HANDLE WINAPI CreateIoCompletionPort(
  _In_     HANDLE    FileHandle,
  _In_opt_ HANDLE    ExistingCompletionPort,
  _In_     ULONG_PTR CompletionKey,
  _In_     DWORD     NumberOfConcurrentThreads
);

BOOL WINAPI GetQueuedCompletionStatus(
  _In_  HANDLE       CompletionPort,
  _Out_ LPDWORD      lpNumberOfBytes,
  _Out_ PULONG_PTR   lpCompletionKey,
  _Out_ LPOVERLAPPED *lpOverlapped,
  _In_  DWORD        dwMilliseconds
);
```

Event-loop thread

1. Prepare a watching set of fds with interested event.
2. Wait on that set.
3. Retrieve events after waiting finish, loop to process events.
4. During processing, modify the watching set maybe, then go back to step 2.

Cross-platform event I/O

- Libuv #ifdef for you so you don't have to.
- Libuv handles weird corner-cases.
- Libuv supports regular file via a fixed size thread pool.
- Libuv use windows API, and correctly handle UTF-16 conversion, even emulate TTY's ansi control codes!

Scale the event loop

- Use multiple threads, each one runs its own event loop.
- Use fork() to pre-fork multiple worker process.
- Use multiple threads, but only one event loop thread, create new worker thread on events.

Take control back

- Wrap callbacks into Promises, Futures...etc, add await, wait... into language.
- Provide real light-weight thread in runtime system, reuse mutex, channel...etc threading concepts.

Meet the GHC RTS

- Running STG model, CMM code.
- Light-weight thread(H-thread) on multiple capability, M:N model.
- Two synchronize primitives: `MVar`, `retry`(in `STM`). Basic atomic operation support.
- Cooperative scheduling via `yield`, Preemptive scheduling via hijack `HpLIM`.
- Moving garbage collection, with basic stable pointer support.

Combine RTS with event loop

- When a lightweight thread encounter a blocking I/O call, allocate an ID, issue event registration, then put the H-thread into background.
- Find proper opportunity to do event polling.
- After polling, get fired IDs back, process them by resuming associated lightweight thread.

How does MIO do it?

- Use fd as the ID. Save a STG callback into an `IntTable`, the callback is just a `putMVar/writeTVar`.
- After event registration, block the H-thread by `takeMVar/retry`, now the H-thread is removed from run-queue.
- An I/O manager H-thread runs alongside with all the user's H-threads on the same capability. Executes a zero timeout poll or forever poll depend on previous result (If successive two polls return no events, we just enter forever poll directly).
- After poll returns, use event's fd as ID to find registered STG callback, execute them, unblocking the H-thread.

Libuv’s stream API

```
int uv_read_start(uv_stream_t* stream
                 , uv_alloc_cb alloc_cb
                 , uv_read_cb read_cb)
void (*uv_read_cb)(uv_stream_t* stream
                  , ssize_t nread
                  , const uv_buf_t* buf)
void (*uv_alloc_cb)(uv_handle_t* handle
                   , size_t suggested_size
                   , uv_buf_t* buf)

int uv_read_stop(uv_stream_t*)

int uv_tcp_init(uv_loop_t* loop, uv_tcp_t* handle)
int uv_run(uv_loop_t* loop, uv_run_mode mode)
typedef enum {
    UV_RUN_DEFAULT = 0,
    UV_RUN_ONCE,
    UV_RUN_NOWAIT
} uv_run_mode;
```

Pass context with data field

```
void* uv_loop_t.data

typedef struct {
   size_t    event_counter;
   size_t*   event_queue;
   char**    buffer_table;
   ssize_t*  buffer_size_table;
} hs_loop_data;


// uv_handle_t is the base type for uv_stream_t, etc.
void* uv_handle_t.data
```

Pass context with data field

- Core data structure in each I/O manager thread:

```
data UVManager = UVManager
    -- The parking lot
    { uvmBlockTable  :: IORef (UnliftedArray (MVar ()))
    -- uv_loop_t*
    , uvmLoop        :: Ptr UVLoop
    -- uv_loop_t->data
    , uvmLoopData    :: Ptr UVLoopData
    ...
    }
```

How does stdio do it?

- Allocate our own integer UVSlot as ID. Hold a growable global MVar table per I/O manager indexed by slot (the parking lot).

- When new I/O handle is initialized, allocated a slot, poke the slot to the `uv_handle_t`'s data field. the MVar under slot's index is used to pause/resume I/O H-thread (the parking spot).

- After event registration, block the H-thread by `takeMVar`.

How does stdio allocate slot?

- A Haskell version

```
data UVManager = UVManager
    { ...
    , uvmFreeSlotList :: MVar [UVSlot]
    ...
    }

allocSlot :: HasCallStack => UVManager -> IO UVSlot
allocSlot (UVManager blockTableRef freeSlotList loop ...) =
    modifyMVar freeSlotList $ \ freeList -> case freeList of
        (s:ss) -> return (ss, s)
        []     -> -- double the blockTable and loop's data

freeSlot :: UVManager -> UVSlot -> IO ()
freeSlot (UVManager _ freeSlotList _ _ _ _ _ _) slot =
    modifyMVar_ freeSlotList $ \ freeList -> return (slot:freeList)
```

How does stdio prepare buffer?

- Before registering I/O read, allocate buffers in Haskell, poke them into the loop's buffer table.

```
peekUVBufferTable :: Ptr UVLoopData
                  -> IO (Ptr (Ptr Word8), Ptr CSsize)
peekUVBufferTable p = (,)
    <$> (#{peek hs_loop_data, buffer_table           } p)
    <*> (#{peek hs_loop_data, buffer_size_table      } p)

pokeBufferTable :: UVManager
                -> UVSlot -> Ptr Word8 -> Int -> IO ()
pokeBufferTable uvm slot buf bufSiz = do
    (bufTable, bufSizTable) <- peekUVBufferTable (uvmLoopData uvm)
    pokeElemOff bufTable slot buf
    pokeElemOff bufSizTable slot (fromIntegral bufSiz)
```

How does libuv side work?

```
void hs_alloc_cb(uv_handle_t* handle, size_t suggested_size, uv_buf_t* buf){
    size_t slot = (size_t)handle->data;
    hs_loop_data* loop_data = handle->loop->data;
    buf->base = loop_data->buffer_table[slot];
    buf->len = loop_data->buffer_size_table[slot];
}
void hs_read_cb (uv_stream_t* stream, ssize_t nread, const uv_buf_t* buf){
    size_t slot = (size_t)stream->data;
    hs_loop_data* loop_data = stream->loop->data;
    if (nread != 0) {
        loop_data->buffer_size_table[slot] = nread;
        loop_data->event_queue[loop_data->event_counter] = slot;
        loop_data->event_counter += 1;
        uv_read_stop(stream);
    }
}
int hs_uv_read_start(uv_stream_t* stream){
    return uv_read_start(stream, hs_alloc_cb, hs_read_cb);
}
```

How does stdio do it?

- I/O manager's polling strategy is similar to MIO's one.

```
step :: UVManager -> Bool -> IO CSize
step (UVManager blockTableRef freeSlotList loop loopData _ _ timer _) block =
    do      blockTable <- readIORef blockTableRef
            clearUVEventCounter loopData         -- clean event counter
            if block
            then if rtsSupportsBoundThreads
                then uvRunSafe loop uV_RUN_ONCE -- forever poll
                else do
                    uvTimerWakeStart timer 2 -- 2ms timer on non-threaded rts
                    uvRun loop uV_RUN_ONCE
            else uvRun loop uV_RUN_NOWAIT    -- zero timeout poll

            (c, q) <- peekUVEventQueue loopData
            forM_ [0..(fromIntegral c-1)] $ \ i -> do
                slot <- peekElemOff q i
                lock <- indexArrM blockTable (fromIntegral slot)
                tryPutMVar lock ()   -- unlock ghc thread with MVar
            return c
```

How does stdio do it?

- After user's H-thread get resumed, peek the buffer_size table to get result.
- We apply the similar procedure to write, so each `uv_stream_t` got two slots.

```
data UVStream = UVStream
    { uvsHandle     :: {-# UNPACK #-} !(Ptr UVHandle)
    , uvsReadSlot   :: {-# UNPACK #-} !UVSlot
    , uvsWriteReq   :: {-# UNPACK #-} !(Ptr UVReq)
    , uvsWriteSlot  :: {-# UNPACK #-} !UVSlot
    , uvsManager    :: UVManager
    }
```

How does stdio do it?

```
readInput :: HasCallStack
          => UVStream -> Ptr Word8 ->  Int -> IO Int
readInput (UVStream handle rslot _ _ uvm) buf len = do
        m <- getBlockMVar uvm rslot
        withUVManager' uvm $ do
            tryTakeMVar m
            pokeBufferTable uvm rslot buf len
            uvReadStart handle
        takeMVar m
        r <- peekBufferTable uvm rslot
        if  | r > 0  -> return r
            | r == fromIntegral uV_EOF -> return 0
            | r < 0 ->  throwUVIfMinus (return r)
```

How does stdio do it?

```
writeOutput :: HasCallStack
            => UVStream -> Ptr Word8 -> Int -> IO ()
writeOutput (UVStream handle _ req wslot uvm) buf len =
    do  m <- getBlockMVar uvm wslot
        withUVManager' uvm $ do
            tryTakeMVar m
            pokeBufferTable uvm wslot buf len
            uvWrite req handle
        takeMVar m
        throwUVIfMinus_ $ peekBufferTable uvm wslot
```

Wait, libuv is not thread-safe!

- You can block threadA with `epoll_wait`, add more events with `epoll_ctl` in threadB, still get your notification.
- But libuv maintain its own data structure, such as callback table, you can't do registration while another thread is still blocking on `uv_run`, which is the case for an `uvRunSafe loop uV_RUN_ONCE`.
- Libuv provides a thread-safe wake up mechanism:

```
int uv_async_init(uv_loop_t* loop, uv_async_t* async
                 , uv_async_cb async_cb)
int uv_async_send(uv_async_t* async)
```

Wait, libuv is not thread-safe!

```
data UVManager = UVManager
    { ...
    , uvmRunning      :: MVar Bool
    , uvmAsync        :: Ptr UVHandle
    ...
    }

withUVManager :: HasCallStack
              => UVManager -> (Ptr UVLoop -> IO a) -> IO a
withUVManager uvm f = do
    r <- withMVar (uvmRunning uvm) $ \ running ->
        if running
        then do uvAsyncSend (uvmcAsync uvm)
                return Nothing
        else do r <- f (uvmLoop uvm)
                return (Just r)
    case r of Just r' -> return r'
              _       -> yield >> withUVManager uvm f
```

Wait, libuv is not thread-safe!

```
startUVManager :: HasCallStack => UVManager -> IO ()
startUVManager uvm@(UVManager _ _ _ running _ _ _) = loop
  where
    loop = do
        e <- withMVar running $ \ _ -> step uvm False
        if e > 0
        then yield >> loop
        else do
            yield
            e <- withMVar running $ \ _ -> step uvm False
            if e > 0 then yield >> loop
            else do
                _ <- swapMVar running True
                _ <- step uvm True
                _ <- swapMVar running False
                yield
                loop
```

OK, then. Is this lock expensive?

- Under load, there should be seldom chances to do safe block poll.
- Handle is bound to a certain `uv_loop_t/UVManager`, we should stop user's thread from migration, e.g. use `forkOn` instead of `forkIO` in server's accept loop.
- Without contention, locks perform just the same as a memory read & write.

Benchmark setup

- Golang vs Node.js Cluster vs MIO vs stdio
- A small server serve 500 bytes in HTTP protocol
- Work load generator: 64 core xeon/128 G ram/10Gb NIC
- Server: 48 core xeon/256G ram/10Gb NIC
- Running wrk with -t64 and -c from 100 to 10000

So, show me the figures!

Things we tried

- Use `StablePtr PrimMVar` as ID, directly resume users' H-thread in C with `hs_try_putmvar`.

- Stable pointers are slow to allocated under contention, and make GC slow even on minor ones!

- The implementation of `hs_try_putmvar` asks for malloc/free of a `PutMVar` struct each invocation, which bring extra overhead.

Things we tried

- Use `STM` to notify users' H-thread `uv_run` is finished.

```
data UVMState = UVM_LOCKED | UVM_POLLING | UVM_FREE
data UVManager = UVManager {
    ...
    ,  uvmState     :: TVar UVMState
    ...
    }

uvAsyncSendSTM :: Ptr UVHandle -> STM ()
uvAsyncSendSTM = unsafeIOToSTM . uvAsyncSend

withUVManager :: HasCallStack => UVManager -> (Ptr UVLoop -> IO a) -> IO a
withUVManager uvm f = bracket_
    (atomically $ do
        state <- readTVar (uvmState uvm)
        case state of
            UVM_LOCKED -> retry
            UVM_POLLING -> uvAsyncSendSTM (uvmAsync uvm) >> retry
            UVM_FREE -> swapTVar (uvmState uvm) UVM_LOCKED)
    (atomically $ swapTVar (uvmState uvm) UVM_FREE)
    (f (uvmLoop uvm))
```

Things we tried

- Allocate slot in C

```
typedef struct {
    ...
    size_t*   slot_table;
    size_t    free_slot;
    size_t    size;
} hs_loop_data;

size_t alloc_slot(uv_loop_t* loop){
    hs_loop_data* loop_data = loop->data;
    size_t r = loop_data->free_slot;
    loop_data->free_slot = loop_data->slot_table[r];
    if (r == loop_data->size - 1) {
         hs_uv_loop_resize(loop, (loop_data->size) * 2);
    }
    return r;
}
void free_slot(uv_loop_t* loop, size_t slot){
    hs_loop_data* loop_data = loop->data;
    loop_data->slot_table[slot] = loop_data->free_slot;
    loop_data->free_slot = slot;
}
```

Future work

- Implement all libuv's I/O operation.
- Improve packed data structures, including text.
- TLS, HTTP protocol, etc.
- Improve stable pointer implementation, and use it.
- Combine H-thread pause/resume with RTS directly.