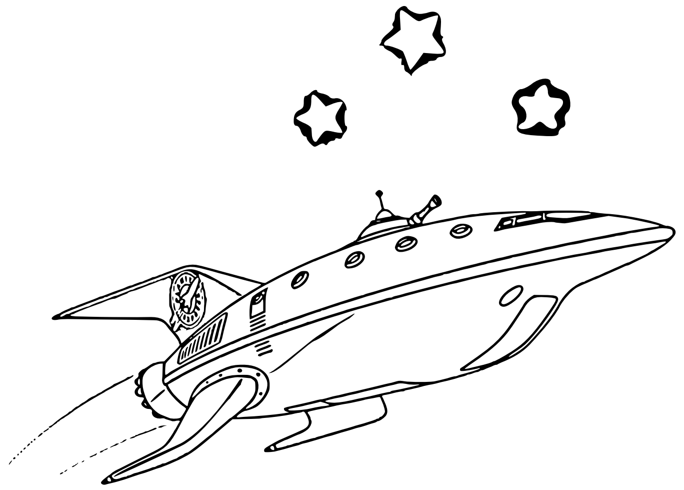


Баукин Антон

Node.js

как система кооперативной многозадачности

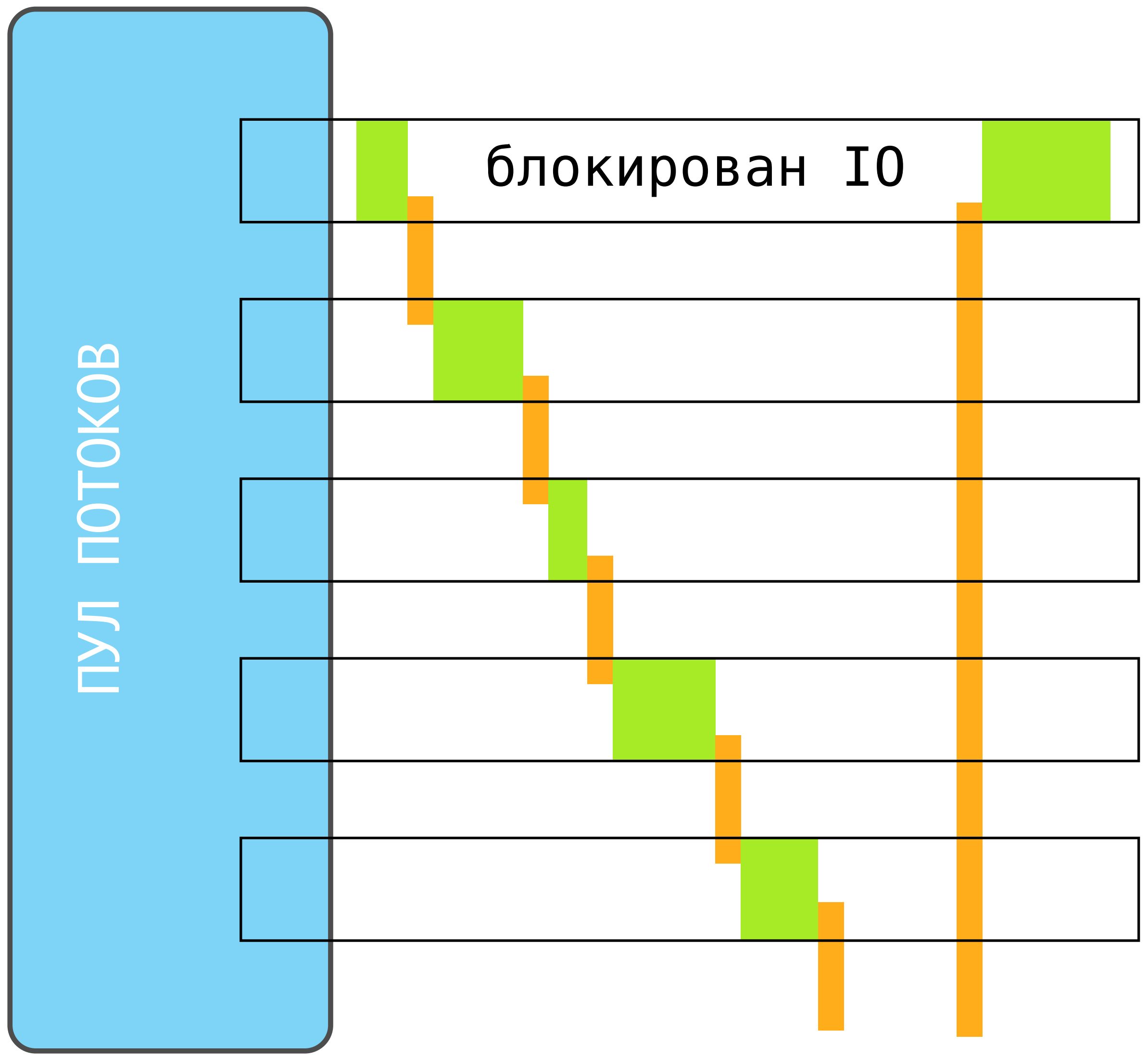
TiTConf

titconf.ru/node.pdf

Copy

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
  char buf[64];
  long done = 0;

  while(1)
  {
    int size = read(0, buf, sizeof(buf));

    if(size == -1) exit(1);
    if(size ==  0) break;

    write(1, buf, size);
    done += size;
  }

  fprintf(stderr, "done: %li bytes\n", done);
}
```

CPU Vs IO



CPU Cost

L1	3
L2	15
RAM	250
DISK	41M
NET	240M

Async 0

```c
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define IOBS 100

typedef struct iob
{
  struct iob* next;
  int         read;  //: buffer read pos
  int         write; //: buffer write pos
  char        buf[IOBS];
} iob;

typedef struct
{
  int  iof;   //: 1 read 2 write done,
              //  8 has bytes read
  int  fdi;   //: input file
  int  fdo;   //: output file
  iob* read;  //: read and write pointers
  iob* write; //  of the same queue
  int  cycles;
  long done;
} job;

#define job_is_done(j)    ((j->iof & 3) == 3)
#define job_has_read(j)   ((j->iof & 1) == 0)
#define job_done_read(j)  j->iof |= 1
#define job_has_write(j)  ((j->iof & 2) == 0)
#define job_done_write(j) j->iof |= 2

job* job_create(int fdi, int fdo)
{
  job* j = (job*) calloc(1, sizeof(job));

  j->fdi = fdi;
  j->fdo = fdo;

  j->read = j->write =
    (iob*) calloc(1, sizeof(iob));

  return j;
}

void job_close(job* j)
{
  close(j->fdi);
  close(j->fdo);

  while(j->write)
  {
    iob* p = j->write->next;
    free(j->write);
    j->write = p;
  }

  free(j);
}

void job_read(job* j)
{
  int s;

  //?: {current buffer is full}
  if(j->read->read == IOBS)
  {
    iob* p = (iob*)
      calloc(1, sizeof(iob));

    j->read->next = p;
    j->read = p;
  }

  s = read(j->fdi,
    j->read->buf + j->read->read,
    IOBS - j->read->read
  );

  if(s == 0) //?: {stream is done}
  {
    if(j->iof & 8)
      job_done_read(j);
  }
  else if(s == -1) //?: {io-error}
  {
    if(errno != EAGAIN)
      exit(1);
  }
  else
  {
    j->read->read += s;
```

Async 1

```c
    j->iof |= 8;
  }
}

void job_write(job* j)
{
  int s;

  //?: {current buffer done}
  if(j->write->write == IOBS)
  {
    //?: {has no more data yet}
    if(!j->write->next)
    {
      if(!job_has_read(j))
        job_done_write(j);
      return;
    }

    iob* p = j->write;
    j->write = j->write->next;

    free(p);
  }

  //?: {has no more data yet}
  if(j->write->write == j->write->read)
  {
    if(!job_has_read(j))
      job_done_write(j);
    return;
  }

  s = write(j->fdo,
    j->write->buf  + j->write->write,
    j->write->read - j->write->write
  );

  if(s == -1) //?: {io-error}
  {
    if(errno != EAGAIN)
      exit(2);
  }

  j->write->write += s;
  j->done += s;
}
```

```c

void job_do(job* j)
{
  j->cycles++;

  if(job_has_read(j))
    job_read(j);

  if(job_has_write(j))
    job_write(j);
}


#define RFLAGS (O_RDONLY | O_NONBLOCK)
#define WFLAGS (O_WRONLY | O_CREAT | \
                O_TRUNC  | O_NONBLOCK)

int main(int argc, char* argv[])
{
  job* j; int fdi, fdo;

  //~: open input-output files
  if(argc != 3) exit(11);
  fdi = open(argv[1], RFLAGS);
  if(fdi == -1) exit(12);
  fdo = open(argv[2], WFLAGS, 0600);
  if(fdo == -1) exit(13);

  //~: open copy job
  j = job_create(fdi, fdo);

  while(!job_is_done(j))
    job_do(j);

  fprintf(stderr, "async: %li bytes"
    " in %i cycles\n", j->done, j->cycles);

  job_close(j);
}
```

```
$ mkfifo queue
$ ./async queue res.html &
$ curl -s -L --limit-rate 5K www.google.ru >queue

async: 18452 bytes in 102,025,933 cycles
```

Epoll 0

```c
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/epoll.h>

#define IOBS 100

typedef struct iob
{
  struct iob* next;
  int         read;  //: buffer read pos
  int         write; //: buffer write pos
  char        buf[IOBS];
} iob;

typedef struct
{
  int  ectl;   //: epoll descriptor
  int  events; //: listeners number
  int  cycles;
} srv;

struct job;
typedef struct
{
  struct job* job;
  int         io;
} jobp;

typedef struct job
{
  srv* server;
  int  ectl;   //: epoll descriptor
  int  iof;    //: 1 read, 2 write done
  int  fdi;    //: input file
  int  fdo;    //: output file
  iob* read;   //: read and write pointers
  iob* write;  //  of the same queue
  long done;

  jobp ijob;   //: typed job pointers
  jobp ojob;
} job;

#define job_is_done(j)   ((j->iof & 3) == 3)
#define job_has_read(j)  ((j->iof & 1) == 0)
#define job_has_write(j) ((j->iof & 2) == 0)

#define IO_IN    EPOLLIN
#define IO_OUT   EPOLLOUT
#define CMD_ADD  EPOLL_CTL_ADD
#define CMD_DEL  EPOLL_CTL_DEL

void job_listen(job* j, int io, int cmd);

job* job_create(int fdi, int fdo)
{
  job* j; struct epoll_event e; int x;
  j = (job*) calloc(1, sizeof(job));

  j->fdi = fdi;
  j->fdo = fdo;

  j->read = j->write =
    (iob*) calloc(1, sizeof(iob));

  j->ijob.job = j->ojob.job = j;
  j->ijob.io  = IO_IN;
  j->ojob.io  = IO_OUT;

  return j;
}

void job_close(job* j)
{
  close(j->fdi);
  close(j->fdo);

  while(j->write)
  {
    iob* p = j->write->next;
    free(j->write);
    j->write = p;
  }

  printf("epoll: %li bytes\n", j->done);
  free(j);
}

void job_write_on(job* j)
{
```

Epoll 1

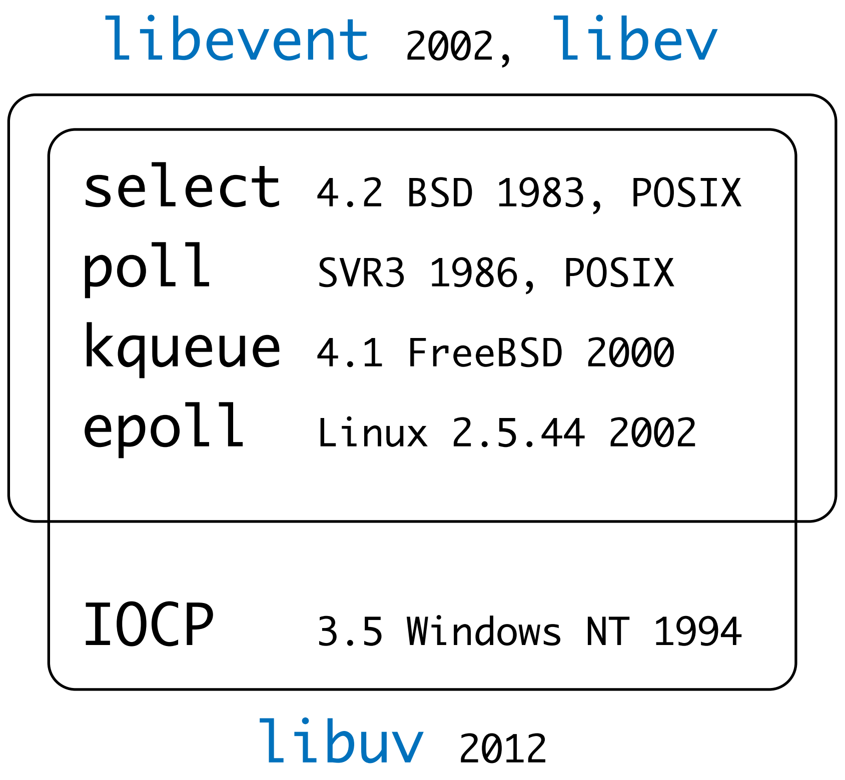
```c
  if(j->iof & 4) return;
  j->iof |= 4;

  job_listen(j, IO_OUT, CMD_ADD);
}

void job_write_off(job* j)
{
  if(!(j->iof & 4)) return;
  j->iof &= ~4;

  job_listen(j, IO_OUT, CMD_DEL);
}

void job_done_read(job* j)
{
  j->iof |= 1;
  job_listen(j, IO_IN, CMD_DEL);
}

void job_done_write(job* j)
{
  j->iof |= 2;
  job_write_off(j);
}

void job_read(job* j)
{
  int s;

  //?: {current buffer is full}
  if(j->read->read == IOBS)
  {
    iob* p = (iob*)
      calloc(1, sizeof(iob));

    j->read->next = p;
    j->read = p;
  }

  s = read(j->fdi,
    j->read->buf + j->read->read,
    IOBS - j->read->read
  );

  if(s == 0)        //?: {stream is done}
    job_done_read(j);
  else if(s == -1) //?: {io-error}
    exit(1);
  else
    j->read->read += s;

  //~: listen write stream
  job_write_on(j);
}

void job_write(job* j)
{
  int s;

  //?: {current buffer done}
  if(j->write->write == IOBS)
  {
    //?: {has no more data yet}
    if(!j->write->next)
    {
      if(!job_has_read(j))
        job_done_write(j);

      job_write_off(j);
      return;
    }

    iob* p = j->write;
    j->write = j->write->next;

    free(p);
  }

  //?: {has no more data yet}
  if(j->write->write == j->write->read)
  {
    if(!job_has_read(j))
      job_done_write(j);

    job_write_off(j);
    return;
  }

  s = write(j->fdo,
    j->write->buf  + j->write->write,
    j->write->read - j->write->write
  );

```

Epoll 2

```c
  if(s == -1) //?: {io-error}
    exit(2);

  j->write->write += s;
  j->done += s;
}

#define RFLAGS O_RDONLY | O_NONBLOCK
#define WFLAGS O_WRONLY | O_NONBLOCK

job* job_open(char* fi, char* fo)
{
  job* j; int fdi, fdo;

  fdi = open(fi, RFLAGS);
  if(fdi == -1) exit(12);
  fdo = open(fo, WFLAGS);
  if(fdo == -1) exit(13);

  //~: open a job
  j = job_create(fdi, fdo);

  return j;
}

void job_attach(srv* s, job* j)
{
  j->server = s;

  //~: listen read stream
  job_listen(j, IO_IN, CMD_ADD);
}

void job_listen(job* j, int io, int cmd)
{
  int x, ex, fd; jobp* jp;
  struct epoll_event e, *ep = &e;

  if(io == IO_OUT)   //?: {write}
  {
    fd = j->fdo;
    jp = &(j->ojob);
    ex = (cmd == CMD_ADD)?(21):(22);
  }
  else               //~: {read}
  {
    fd = j->fdi;
    jp = &(j->ijob);
    ex = (cmd == CMD_ADD)?(23):(24);
  }

  if(cmd == CMD_ADD) //?: {add}
  {
    e.data.ptr = jp;
    e.events   = io;
    j->server->events++;
  }
  else               //~: {remove}
  {
    ep = 0;
    j->server->events--;
  }

  x = epoll_ctl(j->server->ectl, cmd, fd, ep);
  if(x != 0) exit(ex);
}

srv* srv_create()
{
  srv* s = (srv*) calloc(1, sizeof(srv));

  //~: create epoll object
  s->ectl = epoll_create(2);

  return s;
}

void srv_free(srv* s)
{
  //~: close epoll object
  close(s->ectl);

  printf("epoll: %i cycles\n", s->cycles);
  free(s);
}

void srv_cycles(srv* s)
{
  struct epoll_event evs[2];

  while(s->events)
  {
    int i, en = epoll_wait(s->ectl, evs, 2, -1);
    if(en == 0) exit(25);
```

Epoll 3, libevent, libev, libuv

```c

      for(i = 0;(i < en);i++)
      {
        jobp* jp = (jobp*) evs[i].data.ptr;

        if(jp->io == IO_IN)
          job_read(jp->job);

        if(jp->io == IO_OUT)
          job_write(jp->job);

        if(job_is_done(jp->job))
          job_close(jp->job);
      }

      s->cycles++;
    }
}

int main(int argc, char* argv[])
{
  srv* s; job* j;

  //~: open a copy job
  if(argc != 3) exit(11);
  j = job_open(argv[1], argv[2]);

  s = srv_create();
  job_attach(s, j);

  //c: server cycles
  srv_cycles(s);

  srv_free(s);
}
```



```
$ mkfifo queue-in queue-out
$ cat <queue-out >res.html &
$ ./epoll queue-in queue-out &
$ curl -s -L --limit-rate 5K www.google.ru >queue-in

epoll: 18492 bytes
epoll: 198 cycles
```

UV 0

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <uv.h>

#define IOBS 100 /* buffer size */

typedef struct
{
  uv_loop_t *loop;  //: uv-loop
  int        tasks; //: jobs number
  int        cycles;
  uv_idle_t  idle;  //: idle task

} srv;

typedef struct
{
  srv     *server;
  int      fdi;  //: file descriptors
  int      fdo;
  long     done;
  uv_fs_t  oi;   //: uv file requests
  uv_fs_t  oo;   //   for open file
  uv_fs_t  ri;   //: uv file requests
  uv_fs_t  ro;   //   for read-write
  char    *ifile;
  char    *ofile;
  uv_buf_t iob;
  char     buf[IOBS];

} job;


void job_close(job *j);

int  job_close_if(uv_fs_t *req)
{
  job *j = (job*) req->data;

  //?: {has bytes}
  if(req->result >= 0)
    return 0;

  if(req->result < 0) //?: {error}
    fprintf(stderr, "uv io-error: %s\n",
      uv_strerror((int) req->result));

  job_close(j);
  return 1;
}

void job_read_cb(uv_fs_t *ri);

void job_write_cb(uv_fs_t *ro)
{
  job *j = (job*) ro->data;
  fprintf(stderr, "write %s ", ro->result);

  //?: {not able | error}
  if(job_close_if(ro)) return;

  j->done += ro->result;

  //~: read bytes
  j->iob.len = sizeof(j->buf);

  uv_fs_read(j->server->loop, &(j->ri),
    j->fdi, &(j->iob), 1, -1, job_read_cb);
}

void job_read_cb(uv_fs_t *ri)
{
  job *j = (job*) ri->data;
  fprintf(stderr, "read %s ", ri->result);

  //?: {done | error}
  if(job_close_if(ri)) return;

  //~: write bytes
  j->iob.len = ri->result;

  uv_fs_write(j->server->loop, &(j->ro),
    j->fdo, &(j->iob), 1, -1, job_write_cb);
}

#define RFLAGS (O_RDONLY)
#define WFLAGS (O_RDWR | O_CREAT)

void job_openo_cb(uv_fs_t *oo)
{
  job *j = (job*) oo->data;

```

UV 1

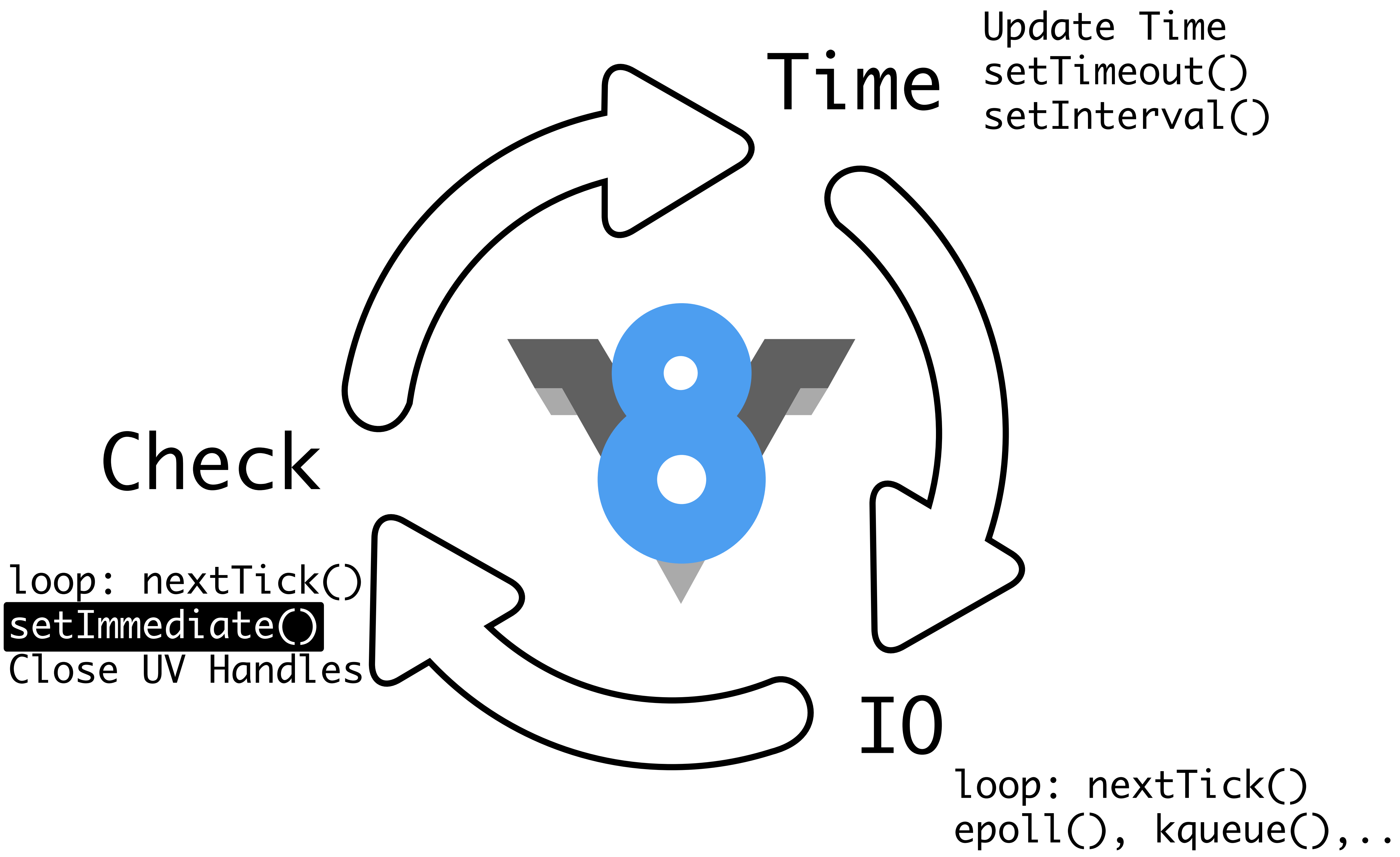
```c
  //~: output file
  j->fdo = oo->result;
  if(j->fdo < 0) exit(12);

  //~: request first read
  uv_fs_read(j->server->loop, &(j->ri),
    j->fdi, &(j->iob), 1, -1, job_read_cb);
}

void job_openi_cb(uv_fs_t *oi)
{
  job *j = (job*) oi->data;

  //~: input file
  j->fdi = oi->result;
  if(j->fdi < 0) exit(13);

  //~: open file to write
  uv_fs_open(j->server->loop, &(j->oo),
    j->ofile, WFLAGS, 0600, job_openo_cb);
}

job* job_create(char* ifile, char* ofile)
{
  job *j = (job*) calloc(1, sizeof(job));

  j->ifile = ifile;
  j->ofile = ofile;

  //~: read-write buffer
  j->iob = uv_buf_init(j->buf, IOBS);

  //~: bind requests
  j->oi.data = j->oo.data = j;
  j->ri.data = j->ro.data = j;

  return j;
}

void job_attach(srv *s, job *j)
{
  j->server = s;
  s->tasks++;

  //~: open file to read
  uv_fs_open(s->loop, &(j->oi),
    j->ifile, RFLAGS, 0, job_openi_cb);
}

void job_close_file(job *j, int fd)
{
  uv_fs_t *r = (uv_fs_t*)
    calloc(1, sizeof(uv_fs_t));

  uv_fs_close(j->server->loop, r, fd, 0);
  uv_fs_req_cleanup(r);
  free(r);
}

void job_close(job *j)
{
  //~: cleanup requests
  uv_fs_req_cleanup(&(j->oi));
  uv_fs_req_cleanup(&(j->oo));
  uv_fs_req_cleanup(&(j->ri));
  uv_fs_req_cleanup(&(j->ro));

  //~: close the files
  job_close_file(j, j->fdi);
  job_close_file(j, j->fdo);

  j->server->tasks--;
  free(j);
}

void srv_idle_cb(uv_idle_t *idle)
{
  srv *s = (srv*)(idle->data);

  //?: {server has tasks}
  if(s->tasks)
    s->cycles++;
  else
    uv_idle_stop(idle);
}

srv* srv_create()
{
  srv* s = (srv*) calloc(1, sizeof(srv));

  //~: use default loop
  s->loop = uv_default_loop();

  //~: add & start idle task
```

UV 2, Copy.js

```c
  s->idle.data = s;
  uv_idle_init(s->loop, &(s->idle));
  uv_idle_start(&(s->idle), srv_idle_cb);

  return s;
}

void srv_free(srv* s)
{
  //~: free idle handle
  uv_close((uv_handle_t*) &(s->idle), 0);

  //~: free the loop
  uv_loop_close(s->loop);

  printf("uv: %i cycles\n", s->cycles);
  free(s);
}

void srv_cycles(srv *s)
{
  //~: run uv-loop
  uv_run(s->loop, UV_RUN_DEFAULT);
}

void main(int argc, char* argv[])
{
  //~: create uv-loop
  srv* s; job* j;

  //~: open a copy job
  if(argc != 3) exit(11);
  j = job_create(argv[1], argv[2]);

  //~: create server & attach job
  s = srv_create();
  job_attach(s, j);

  //c: server cycles
  srv_cycles(s);

  srv_free(s);
}
```

```javascript
function copy(ifile, ofile, callback)
{
    var fs = require('fs')
    var fdi, fdo, buf = new Buffer(100)

    fs.open(ifile, 'r', function(err, fd)
    {
        if(err) return done(err); else fdi = fd
        fs.open(ofile, 'w', function(err, fd)
        {
            if(err) return done(err); else fdo = fd
            fs.read(fdi, buf, 0, buf.length, null, onread)
        })
    })

    function done(err)
    {
        if(!fdi) return callback(err)
        fs.close(fdi, function(exx)
        {
            if(!fdo) return callback(err || exx)

            fs.close(fdo, function(eyy)  {
                callback(err || exx || eyy)
            })
        })
    }

    function onread(err, size, buf)
    {
        if(!size || err) return done(err)
        fs.write(fdo, buf, 0, size, onwrite)
    }

    function onwrite(err, size, buf)
    {
        if(err) return done(err)
        fs.read(fdi, buf, 0, buf.length, null, onread)
    }
}
```

Node.js



libuv loop

Cooperative Node.js

```
/* load huge JSON with 10K objects */
var data = require('./database.json')

function count(db, callback)
{
    var i = 0, map = {}, exp = /[a-z]+/gi
    var stat = { rows: 0, words: map }

    function next()
    {
        if(i >= db.length) return done()
        stat.rows++

        var s, x = db[i++]
        if(!x.about) done(new Error(
          'Record [' + (i-1) + '] has no about!'))

        exp.lastIndex = 0
        while(s = exp.exec(x.about))
        {
            s = s[0].toLowerCase()
            map[s] = (map[s])?(map[s] + 1):(1)
        }

        //!: will fire on the next loop
        setImmediate(next)
    }

    function done(err)
    {
        callback(err, (err)?(undefined):(stat))
    }

    next() //<-- first invocation
}

count(data, function(err, stat)
{
    var ks = Object.keys(stat.words)
    ks.sort(); ks.forEach(function(k) {
        var x = k; while(x.length < 20) x += '.'
        console.log(x + stat.words[k])
    })
})
```

Вызов setImmediate() внутри вызванного им callback будет обработан уже в след. цикле!