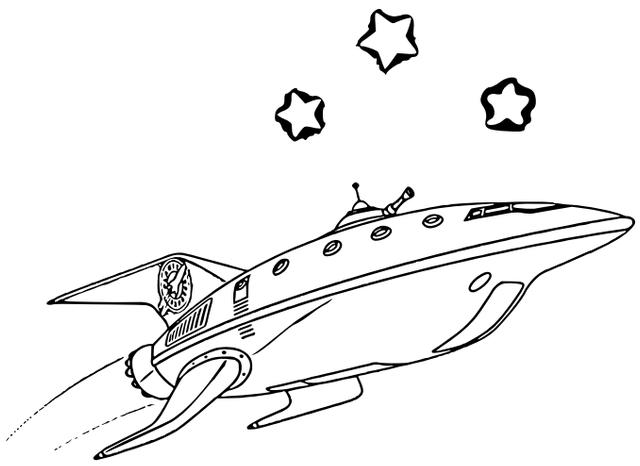


Баукин Антон

Node.js

как система кооперативной многозадачности

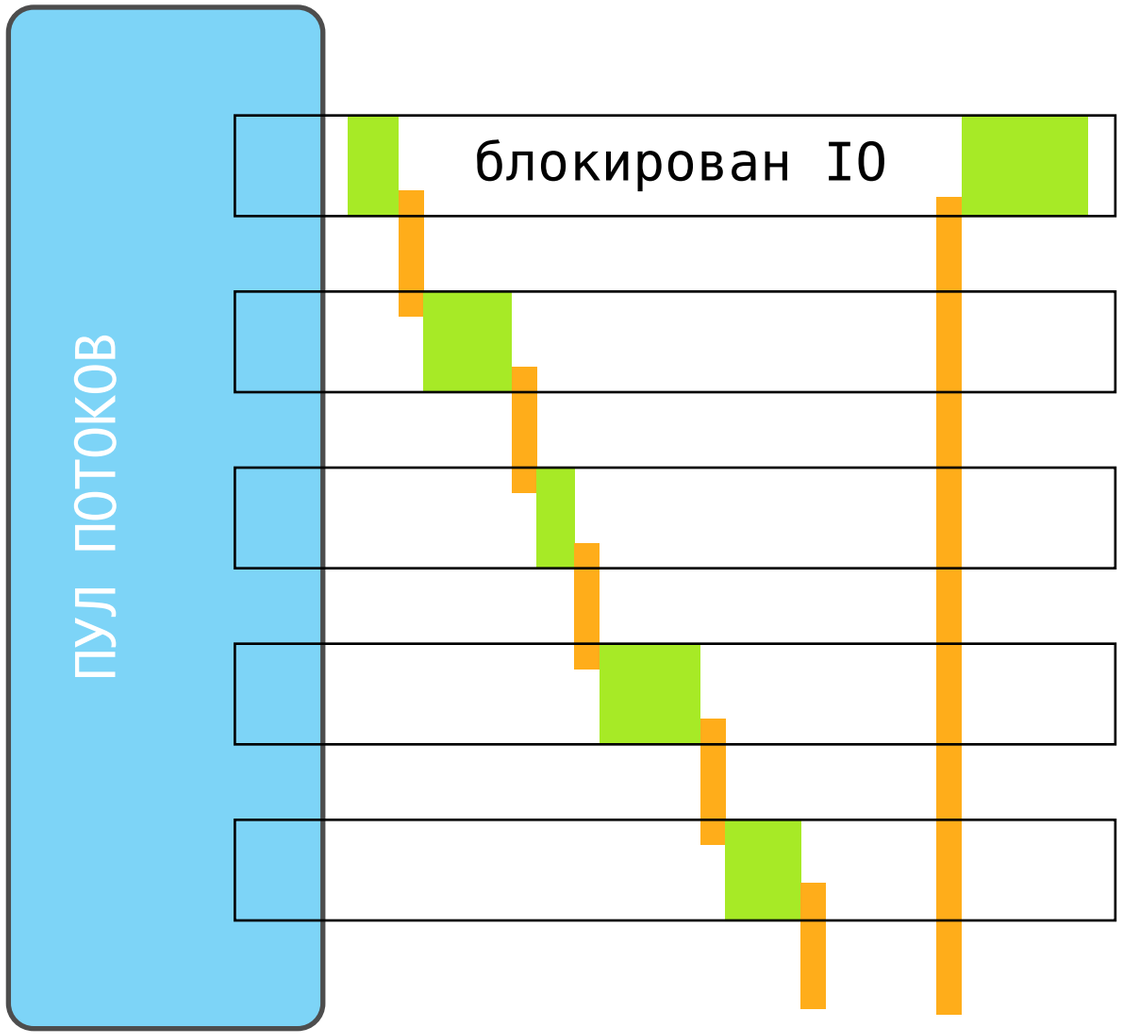
TiTConf

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

```
    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;
}
```

```

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

```
 95    if(j->iof & 4) return;
 96    j->iof |= 4;
 97
 98    job_listen(j, IO_OUT, CMD_ADD);
 99  }
100
101  void job_write_off(job* j)
102  {
103    if(!(j->iof & 4)) return;
104    j->iof &= ~4;
105
106    job_listen(j, IO_OUT, CMD_DEL);
107  }
108
109  void job_done_read(job* j)
110  {
111    j->iof |= 1;
112    job_listen(j, IO_IN, CMD_DEL);
113  }
114
115  void job_done_write(job* j)
116  {
117    j->iof |= 2;
118    job_write_off(j);
119  }
120
121  void job_read(job* j)
122  {
123    int s;
124
125    //?: {current buffer is full}
126    if(j->read->read == IOBS)
127    {
128      iob* p = (iob*)
129        calloc(1, sizeof(iob));
130
131      j->read->next = p;
132      j->read = p;
133    }
134
135    s = read(j->fdi,
136      j->read->buf + j->read->read,
137      IOBS - j->read->read
138    );
139
140    if(s == 0)       //?: {stream is done}
141      job_done_read(j);
142    else if(s == -1) //?: {io-error}
143      exit(1);
144    else
145      j->read->read += s;
146
147    //~: listen write stream
148    job_write_on(j);
149  }
150
151  void job_write(job* j)
152  {
153    int s;
154
155    //?: {current buffer done}
156    if(j->write->write == IOBS)
157    {
158      //?: {has no more data yet}
159      if(!j->write->next)
160      {
161        if(!job_has_read(j))
162          job_done_write(j);
163
164        job_write_off(j);
165        return;
166      }
167
168      iob* p = j->write;
169      j->write = j->write->next;
170
171      free(p);
172    }
173
174    //?: {has no more data yet}
175    if(j->write->write == j->write->read)
176    {
177      if(!job_has_read(j))
178        job_done_write(j);
179
180      job_write_off(j);
181      return;
182    }
183
184    s = write(j->fdo,
185      j->write->buf  + j->write->write,
186      j->write->read - j->write->write
187    );
188
```

Epoll 2

```
189     if(s == -1) //?: {io-error}
190       exit(2);
191
192     j->write->write += s;
193     j->done += s;
194   }
195
196   #define RFLAGS O_RDONLY | O_NONBLOCK
197   #define WFLAGS O_WRONLY | O_NONBLOCK
198
199   job* job_open(char* fi, char* fo)
200   {
201     job* j; int fdi, fdo;
202
203     fdi = open(fi, RFLAGS);
204     if(fdi == -1) exit(12);
205     fdo = open(fo, WFLAGS);
206     if(fdo == -1) exit(13);
207
208     //~: open a job
209     j = job_create(fdi, fdo);
210
211     return j;
212   }
213
214   void job_attach(srv* s, job* j)
215   {
216     j->server = s;
217
218     //~: listen read stream
219     job_listen(j, IO_IN, CMD_ADD);
220   }
221
222   void job_listen(job* j, int io, int cmd)
223   {
224     int x, ex, fd; jobp* jp;
225     struct epoll_event e, *ep = &e;
226
227     if(io == IO_OUT)   //?: {write}
228     {
229       fd = j->fdo;
230       jp = &(j->ojob);
231       ex = (cmd == CMD_ADD)?(21):(22);
232     }
233     else               //~: {read}
234     {
235       fd = j->fdi;
236       jp = &(j->ijob);
237       ex = (cmd == CMD_ADD)?(23):(24);
238     }
239
240     if(cmd == CMD_ADD) //?: {add}
241     {
242       e.data.ptr = jp;
243       e.events   = io;
244       j->server->events++;
245     }
246     else               //~: {remove}
247     {
248       ep = 0;
249       j->server->events--;
250     }
251
252     x = epoll_ctl(j->server->ectl, cmd, fd, ep);
253     if(x != 0) exit(ex);
254   }
255
256   srv* srv_create()
257   {
258     srv* s = (srv*) calloc(1, sizeof(srv));
259
260     //~: create epoll object
261     s->ectl = epoll_create(2);
262
263     return s;
264   }
265
266   void srv_free(srv* s)
267   {
268     //~: close epoll object
269     close(s->ectl);
270
271     printf("epoll: %i cycles\n", s->cycles);
272     free(s);
273   }
274
275   void srv_cycles(srv* s)
276   {
277     struct epoll_event evs[2];
278
279     while(s->events)
280     {
281       int i, en = epoll_wait(s->ectl, evs, 2, -1);
282       if(en == 0) exit(25);
```

Epoll 3, libevent, libev, libuv

```
283
284     for(i = 0;(i < en);i++)
285     {
286       jobp* jp = (jobp*) evs[i].data.ptr;
287
288       if(jp->io == IO_IN)
289         job_read(jp->job);
290
291       if(jp->io == IO_OUT)
292         job_write(jp->job);
293
294       if(job_is_done(jp->job))
295         job_close(jp->job);
296     }
297
298     s->cycles++;
299   }
300 }
301
302 int main(int argc, char* argv[])
303 {
304   srv* s; job* j;
305
306   //~: open a copy job
307   if(argc != 3) exit(11);
308   j = job_open(argv[1], argv[2]);
309
310   s = srv_create();
311   job_attach(s, j);
312
313   //c: server cycles
314   srv_cycles(s);
315
316   srv_free(s);
317 }
```

libevent 2002, libev

select	4.2 BSD 1983, POSIX
poll	SVR3 1986, POSIX
kqueue	4.1 FreeBSD 2000
epoll	Linux 2.5.44 2002
IOCP	3.5 Windows NT 1994

libuv 2012

```
$ 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;

```

```
  //~: 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
```

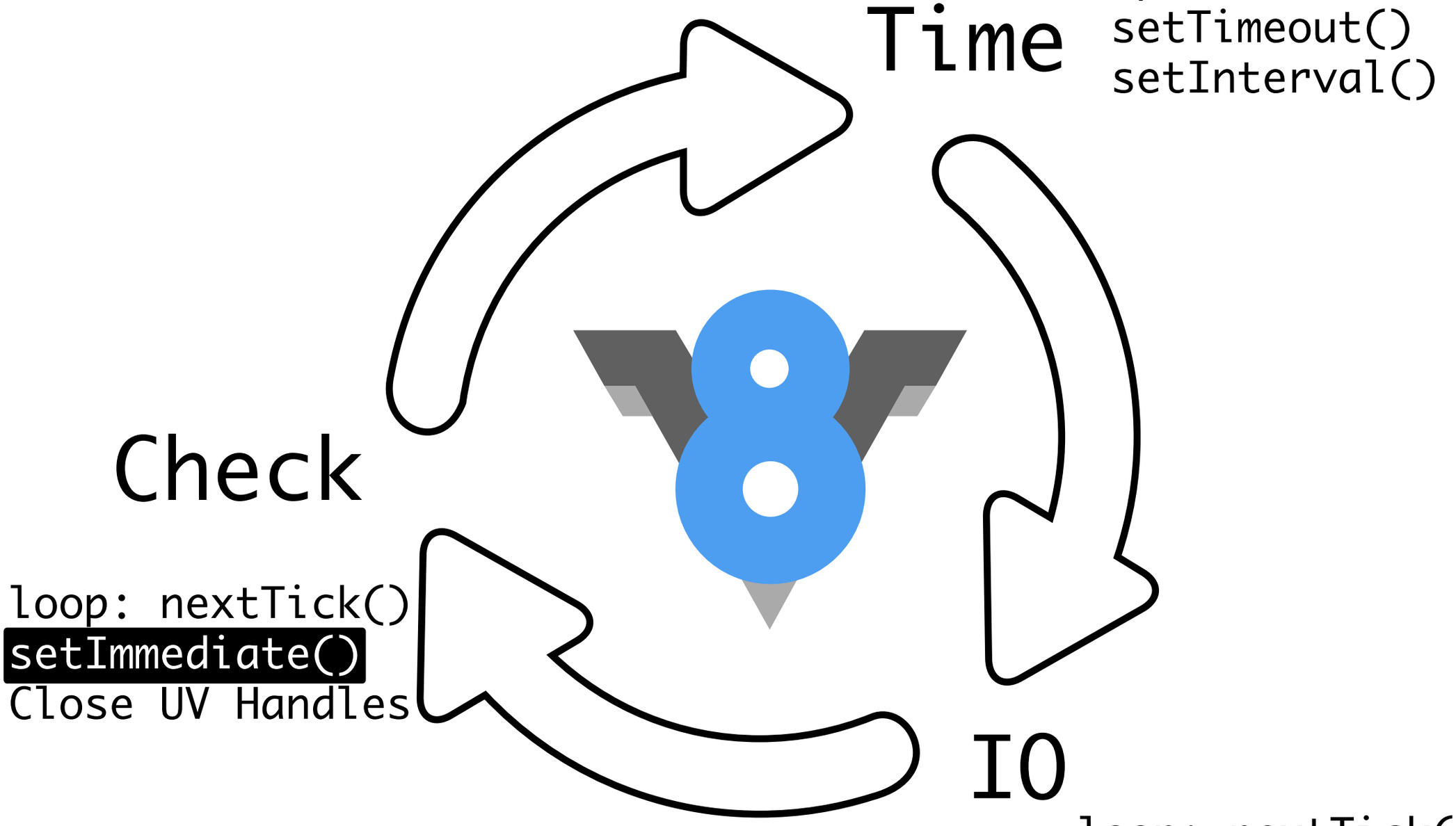
UV 2, Copy.js

```
  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);
}
```

```
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)
    }
}
```


AT&T
3G
9:42 AM
SMS
Text
Tuesday
9
Calendar
Photos
Camera
YouTube
Stocks
280
Maps
73°
Weather
Clock
Calculator
Notes
Settings
iTunes
App Store
Phone
Mail
Safari
iPod

Node.js



libuv loop

Cooperative Node.js

```
 1  /* load huge JSON with 10K objects */
 2  var data = require('./database.json')
 3
 4  function count(db, callback)
 5  {
 6      var i = 0, map = {}, exp = /[a-z]+/gi
 7      var stat = { rows: 0, words: map }
 8
 9      function next()
10      {
11          if(i >= db.length) return done()
12          stat.rows++
13
14          var s, x = db[i++]
15          if(!x.about) done(new Error(
16            'Record [' + (i-1) + '] has no about!'))
17
18          exp.lastIndex = 0
19          while(s = exp.exec(x.about))
20          {
21              s = s[0].toLowerCase()
22              map[s] = (map[s])?(map[s] + 1):(1)
23          }
24
25          //!: will fire on the next loop
26          setImmediate(next)
27      }
28
29      function done(err)
30      {
31          callback(err, (err)?(undefined):(stat))
32      }
33
34      next() //<-- first invocation
35  }
36
37  count(data, function(err, stat)
38  {
39      var ks = Object.keys(stat.words)
40      ks.sort(); ks.forEach(function(k) {
41          var x = k; while(x.length < 20) x += '.'
42          console.log(x + stat.words[k])
43      })
44  })
```

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

Sad but True?

Linux Native POSIX
Thread Library, NPTL

With NPTL on, Sun and Blackwidow JVM 1.4.2 scaled easily to 5000+ threads. Blockin model was consistently 25-35% faster than using NIO selectors. Lot of techniques suggested by EmberIO folks were employed - using multiple selectors, doing multiple (2) reads if the first read returned EAGAIN equivalent in Java. Yet we couldn't beat the plain thread per connection model with Linux NPTL.

Results shows that the cost of NIO selectors coupled with OS polling mechanism (in this case efficient epoll VS selector/poll) has a significant overhead compared to the cost of context switching threads on an NPTL Linux kernel.

Without NPTL of course it's a different story. The blocking server just melts at 400 concurrent connections! We have run the test upto 10K connections and the blocking server outperformed NIO driven selector based server by same margin. Moral of the story - NIO arrives at the scene a little too late - with adequate RAM and better threading models (NPTL), performance gains of NIO don't show up.

June 18 2004