

Баукин Антон

# Node.js

как система кооперативной  
многозадачности

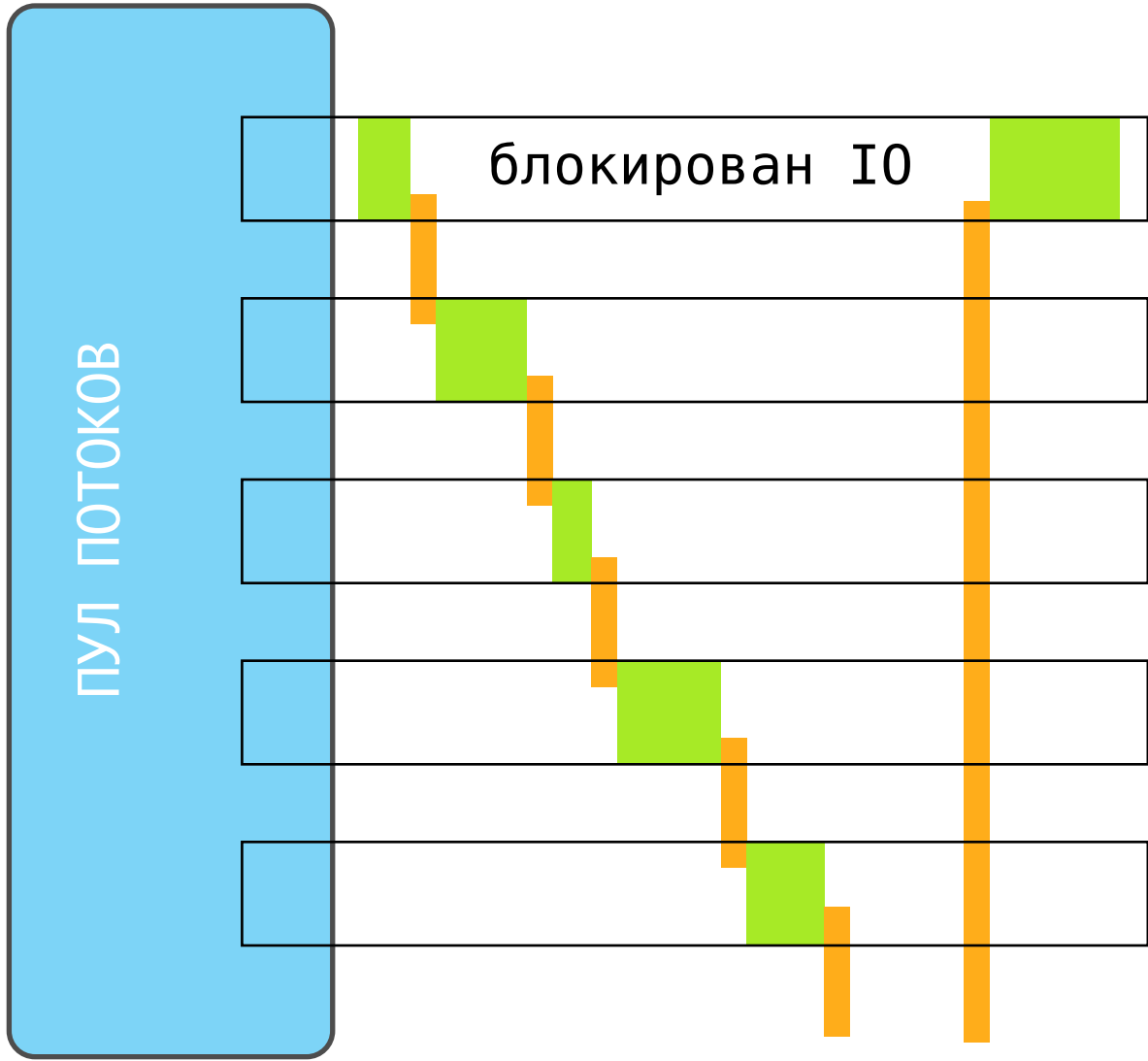
TiTConf

[titconf.ru/node.pdf](http://titconf.ru/node.pdf)

# Copy

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main()
6  {
7      char buf[64];
8      long done = 0;
9
10     while(1)
11     {
12         int size = read(0, buf, sizeof(buf));
13
14         if(size == -1) exit(1);
15         if(size == 0) break;
16
17         write(1, buf, size);
18         done += size;
19     }
20
21     fprintf(stderr, "done: %li bytes\n", done);
22 }
```

CPU Vs IO



| CPU Cost |      |
|----------|------|
| L1       | 3    |
| L2       | 15   |
| RAM      | 250  |
| DISK     | 41M  |
| NET      | 240M |

# Async 0

```
1 #include <errno.h>
2 #include <fcntl.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6
7 #define IOBS 100
8
9 typedef struct iob
10 {
11     struct iob* next;
12     int read;    //: buffer read pos
13     int write;   //: buffer write pos
14     char buf[IOBS];
15 } iob;
16
17 typedef struct
18 {
19     int iof;     //: 1 read 2 write done,
20                // 8 has bytes read
21     int fdi;     //: input file
22     int fdo;     //: output file
23     iob* read;   //: read and write pointers
24     iob* write;  // of the same queue
25     int cycles;
26     long done;
27 } job;
28
29 #define job_is_done(j) ((j->iof & 3) == 3)
30 #define job_has_read(j) ((j->iof & 1) == 0)
31 #define job_done_read(j) j->iof |= 1
32 #define job_has_write(j) ((j->iof & 2) == 0)
33 #define job_done_write(j) j->iof |= 2
34
35 job* job_create(int fdi, int fdo)
36 {
37     job* j = (job*) calloc(1, sizeof(job));
38
39     j->fdi = fdi;
40     j->fdo = fdo;
41
42     j->read = j->write =
43         (iob*) calloc(1, sizeof(iob));
44
45     return j;
46 }
47
```

```
48 void job_close(job* j)
49 {
50     close(j->fdi);
51     close(j->fdo);
52
53     while(j->write)
54     {
55         iob* p = j->write->next;
56         free(j->write);
57         j->write = p;
58     }
59
60     free(j);
61 }
62
63 void job_read(job* j)
64 {
65     int s;
66
67     //?: {current buffer is full}
68     if(j->read->read == IOBS)
69     {
70         iob* p = (iob*)
71             calloc(1, sizeof(iob));
72
73         j->read->next = p;
74         j->read = p;
75     }
76
77     s = read(j->fdi,
78             j->read->buf + j->read->read,
79             IOBS - j->read->read
80 );
81
82     if(s == 0) //?: {stream is done}
83     {
84         if(j->iof & 8)
85             job_done_read(j);
86     }
87     else if(s == -1) //?: {io-error}
88     {
89         if(errno != EAGAIN)
90             exit(1);
91     }
92     else
93     {
94         j->read->read += s;
95     }
96 }
```

# Async 1

```
95     j->iof |= 8;
96 }
97 }
98
99 void job_write(job* j)
100 {
101     int s;
102
103     /*?: {current buffer done}
104     if(j->write->write == IOBS)
105     {
106         /*?: {has no more data yet}
107         if(!j->write->next)
108         {
109             if(!job_has_read(j))
110                 job_done_write(j);
111             return;
112         }
113
114         iob* p = j->write;
115         j->write = j->write->next;
116
117         free(p);
118     }
119
120     /*?: {has no more data yet}
121     if(j->write->write == j->write->read)
122     {
123         if(!job_has_read(j))
124             job_done_write(j);
125         return;
126     }
127
128     s = write(j->fdo,
129             j->write->buf + j->write->write,
130             j->write->read - j->write->write
131             );
132
133     if(s == -1) /*?: {io-error}
134     {
135         if(errno != EAGAIN)
136             exit(2);
137     }
138
139     j->write->write += s;
140     j->done += s;
141 }
```

```
142
143 void job_do(job* j)
144 {
145     j->cycles++;
146
147     if(job_has_read(j))
148         job_read(j);
149
150     if(job_has_write(j))
151         job_write(j);
152 }
153
154 #define RFLAGS (O_RDONLY | O_NONBLOCK)
155 #define WFLAGS (O_WRONLY | O_CREAT | \
156               O_TRUNC | O_NONBLOCK)
157
158 int main(int argc, char* argv[])
159 {
160     job* j; int fdi, fdo;
161
162     /*~: open input-output files
163     if(argc != 3) exit(11);
164     fdi = open(argv[1], RFLAGS);
165     if(fdi == -1) exit(12);
166     fdo = open(argv[2], WFLAGS, 0600);
167     if(fdo == -1) exit(13);
168
169     /*~: open copy job
170     j = job_create(fdi, fdo);
171
172     while(!job_is_done(j))
173         job_do(j);
174
175     fprintf(stderr, "async: %li bytes"
176            " in %i cycles\n", j->done, j->cycles);
177
178     job_close(j);
179 }
180
```

\$ 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

```
1 #include <errno.h>
2 #include <fcntl.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6
7 #include <sys/epoll.h>
8
9 #define IOBS 100
10
11 typedef struct iob
12 {
13     struct iob* next;
14     int read;    ///< buffer read pos
15     int write;  ///< buffer write pos
16     char buf[IOBS];
17 } iob;
18
19 typedef struct
20 {
21     int ectl;    ///< epoll descriptor
22     int events;  ///< listeners number
23     int cycles;
24 } srv;
25
26 struct job;
27 typedef struct
28 {
29     struct job* job;
30     int io;
31 } jobp;
32
33 typedef struct job
34 {
35     srv* server;
36     int ectl;    ///< epoll descriptor
37     int iof;     ///< 1 read, 2 write done
38     int fdi;     ///< input file
39     int fdo;     ///< output file
40     iob* read;   ///< read and write pointers
41     iob* write;  ///< of the same queue
42     long done;
43
44     jobp ijob;   ///< typed job pointers
45     jobp ojob;
46 } job;
47
```

```
48 #define job_is_done(j) ((j->iof & 3) == 3)
49 #define job_has_read(j) ((j->iof & 1) == 0)
50 #define job_has_write(j) ((j->iof & 2) == 0)
51
52 #define IO_IN    EPOLLIN
53 #define IO_OUT   EPOLLOUT
54 #define CMD_ADD  EPOLL_CTL_ADD
55 #define CMD_DEL  EPOLL_CTL_DEL
56
57 void job_listen(job* j, int io, int cmd);
58
59 job* job_create(int fdi, int fdo)
60 {
61     job* j; struct epoll_event e; int x;
62     j = (job*) calloc(1, sizeof(job));
63
64     j->fdi = fdi;
65     j->fdo = fdo;
66
67     j->read = j->write =
68         (iob*) calloc(1, sizeof(iob));
69
70     j->ijob.job = j->ojob.job = j;
71     j->ijob.io = IO_IN;
72     j->ojob.io = IO_OUT;
73
74     return j;
75 }
76
77 void job_close(job* j)
78 {
79     close(j->fdi);
80     close(j->fdo);
81
82     while(j->write)
83     {
84         iob* p = j->write->next;
85         free(j->write);
86         j->write = p;
87     }
88
89     printf("epoll: %li bytes\n", j->done);
90     free(j);
91 }
92
93 void job_write_on(job* j)
94 {
95
```

# 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     //~: server cycles
314     srv_cycles(s);
315
316     srv_free(s);
317 }
```

```
$ 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
```

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

# UV 0

```
1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <uv.h>
6
7 #define IOBS 100 /* buffer size */
8
9 typedef struct
10 {
11     uv_loop_t *loop; /*: uv-loop
12     int tasks; /*: jobs number
13     int cycles;
14     uv_idle_t idle; /*: idle task
15
16 } srv;
17
18 typedef struct
19 {
20     srv *server;
21     int fdi; /*: file descriptors
22     int fdo;
23     long done;
24     uv_fs_t oi; /*: uv file requests
25     uv_fs_t oo; /* for open file
26     uv_fs_t ri; /*: uv file requests
27     uv_fs_t ro; /* for read-write
28     char *ifile;
29     char *ofile;
30     uv_buf_t iob;
31     char buf[IOBS];
32
33 } job;
34
35
36 void job_close(job *j);
37
38 int job_close_if(uv_fs_t *req)
39 {
40     job *j = (job*) req->data;
41
42     /*?: {has bytes}
43     if(req->result >= 0)
44         return 0;
45
46     if(req->result < 0) /*?: {error}
47         fprintf(stderr, "uv io-error: %s\n",
```

```
48         uv_strerror((int) req->result));
49
50     job_close(j);
51     return 1;
52 }
53
54 void job_read_cb(uv_fs_t *ri);
55
56 void job_write_cb(uv_fs_t *ro)
57 {
58     job *j = (job*) ro->data;
59     fprintf(stderr, "write %s ", ro->result);
60
61     /*?: {not able | error}
62     if(job_close_if(ro)) return;
63
64     j->done += ro->result;
65
66     /*~: read bytes
67     j->iob.len = sizeof(j->buf);
68
69     uv_fs_read(j->server->loop, &(j->ri),
70         j->fdi, &(j->iob), 1, -1, job_read_cb);
71 }
72
73 void job_read_cb(uv_fs_t *ri)
74 {
75     job *j = (job*) ri->data;
76     fprintf(stderr, "read %s ", ri->result);
77
78     /*?: {done | error}
79     if(job_close_if(ri)) return;
80
81     /*~: write bytes
82     j->iob.len = ri->result;
83
84     uv_fs_write(j->server->loop, &(j->ro),
85         j->fdo, &(j->iob), 1, -1, job_write_cb);
86 }
87
88 #define RFLAGS (O_RDONLY)
89 #define WFLAGS (O_RDWR | O_CREAT)
90
91 void job_openo_cb(uv_fs_t *oo)
92 {
93     job *j = (job*) oo->data;
94
```

# UV 1

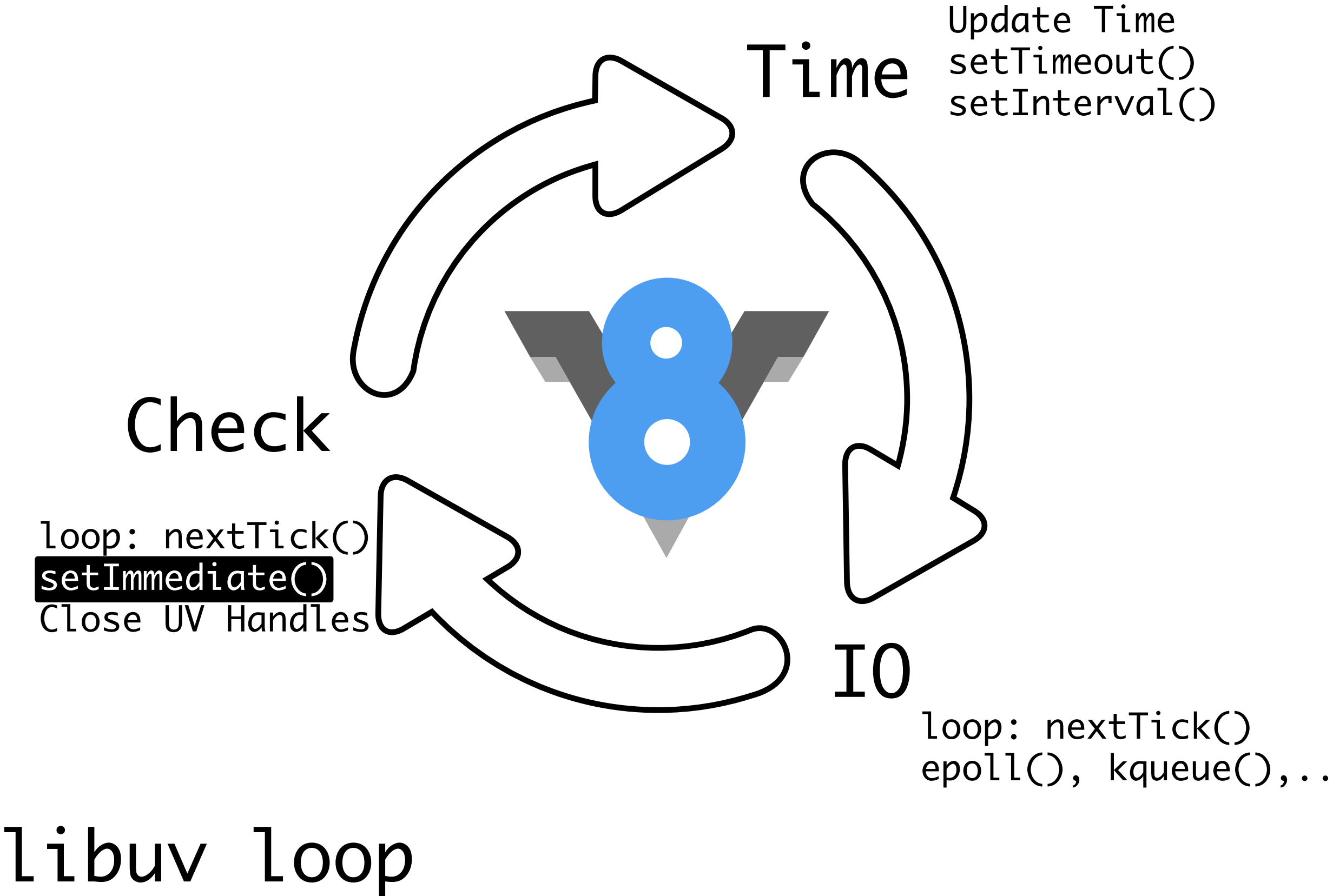
```
95  //~: output file
96  j->fdo = oo->result;
97  if(j->fdo < 0) exit(12);
98
99  //~: request first read
100 uv_fs_read(j->server->loop, &(j->ri),
101           j->fdi, &(j->iob), 1, -1, job_read_cb);
102 }
103
104 void job_openi_cb(uv_fs_t *oi)
105 {
106     job *j = (job*) oi->data;
107
108     //~: input file
109     j->fdi = oi->result;
110     if(j->fdi < 0) exit(13);
111
112     //~: open file to write
113     uv_fs_open(j->server->loop, &(j->oo),
114               j->ofile, WFLAGS, 0600, job_openo_cb);
115 }
116
117 job* job_create(char* ifile, char* ofile)
118 {
119     job *j = (job*) calloc(1, sizeof(job));
120
121     j->ifile = ifile;
122     j->ofile = ofile;
123
124     //~: read-write buffer
125     j->iob = uv_buf_init(j->buf, IOBS);
126
127     //~: bind requests
128     j->oi.data = j->oo.data = j;
129     j->ri.data = j->ro.data = j;
130
131     return j;
132 }
133
134 void job_attach(srv *s, job *j)
135 {
136     j->server = s;
137     s->tasks++;
138
139     //~: open file to read
140     uv_fs_open(s->loop, &(j->oi),
141               j->ifile, RFLAGS, 0, job_openi_cb);
```

```
142 }
143
144 void job_close_file(job *j, int fd)
145 {
146     uv_fs_t *r = (uv_fs_t*)
147         calloc(1, sizeof(uv_fs_t));
148
149     uv_fs_close(j->server->loop, r, fd, 0);
150     uv_fs_req_cleanup(r);
151     free(r);
152 }
153
154 void job_close(job *j)
155 {
156     //~: cleanup requests
157     uv_fs_req_cleanup(&(j->oi));
158     uv_fs_req_cleanup(&(j->oo));
159     uv_fs_req_cleanup(&(j->ri));
160     uv_fs_req_cleanup(&(j->ro));
161
162     //~: close the files
163     job_close_file(j, j->fdi);
164     job_close_file(j, j->fdo);
165
166     j->server->tasks--;
167     free(j);
168 }
169
170 void srv_idle_cb(uv_idle_t *idle)
171 {
172     srv *s = (srv*)(idle->data);
173
174     //?: {server has tasks}
175     if(s->tasks)
176         s->cycles++;
177     else
178         uv_idle_stop(idle);
179 }
180
181 srv* srv_create()
182 {
183     srv* s = (srv*) calloc(1, sizeof(srv));
184
185     //~: use default loop
186     s->loop = uv_default_loop();
187
188     //~: add & start idle task
```

## UV 2, Copy.js

```
189 s->idle.data = s;
190 uv_idle_init(s->loop, &(s->idle));
191 uv_idle_start(&(s->idle), srv_idle_cb);
192
193 return s;
194 }
195
196 void srv_free(srv* s)
197 {
198     //~: free idle handle
199     uv_close((uv_handle_t*) &(s->idle), 0);
200
201     //~: free the loop
202     uv_loop_close(s->loop);
203
204     printf("uv: %i cycles\n", s->cycles);
205     free(s);
206 }
207
208 void srv_cycles(srv *s)
209 {
210     //~: run uv-loop
211     uv_run(s->loop, UV_RUN_DEFAULT);
212 }
213
214 void main(int argc, char* argv[])
215 {
216     //~: create uv-loop
217     srv* s; job* j;
218
219     //~: open a copy job
220     if(argc != 3) exit(11);
221     j = job_create(argv[1], argv[2]);
222
223     //~: create server & attach job
224     s = srv_create();
225     job_attach(s, j);
226
227     //~: server cycles
228     srv_cycles(s);
229
230     srv_free(s);
231 }
```

```
1 function copy(ifile, ofile, callback)
2 {
3     var fs = require('fs')
4     var fdi, fdo, buf = new Buffer(100)
5
6     fs.open(ifile, 'r', function(err, fd)
7     {
8         if(err) return done(err); else fdi = fd
9         fs.open(ofile, 'w', function(err, fd)
10        {
11            if(err) return done(err); else fdo = fd
12            fs.read(fdi, buf, 0, buf.length, null, onread)
13        })
14    })
15
16    function done(err)
17    {
18        if(!fdi) return callback(err)
19        fs.close(fdi, function(exx)
20        {
21            if(!fdo) return callback(err || exx)
22
23            fs.close(fdo, function(eyy) {
24                callback(err || exx || eyy)
25            })
26        })
27    }
28
29    function onread(err, size, buf)
30    {
31        if(!size || err) return done(err)
32        fs.write(fdo, buf, 0, size, onwrite)
33    }
34
35    function onwrite(err, size, buf)
36    {
37        if(err) return done(err)
38        fs.read(fdi, buf, 0, buf.length, null, onread)
39    }
40 }
```



# 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     }
```

```
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!