ПО сетевых устройств

Трещановский Павел Александрович, к.т.н.

06.05.20

Сервер на основе блокирующих вызовов

```
int handle_requests(int data_sock)
{
	while (1) {
		char rx_buf[100];

		ret = recv(data_sock, rx_buf, sizeof(rx_buf), 0);
		if (ret == 0)
			break;

		/* Обработка запроса. */
	}
}

int handle_connections(int listen_sock)
{
	while (1) {
		int data_sock;

		data_sock = accept(listen_sock, NULL, NULL);

		ret = handle_requests(data_sock);
	}
}
```

Замечания

- `recv()` и `accept()` исполняются неопределенное время - до появления новых данных/соединений.
- Сервер не является многозадачным - не поддерживается одновременная обработка из нескольких соединений.
- Многие системные вызовы, отвечающие за ввод-вывод, являются блокирующими: `read()`, `write()`, `open()`, `connect()`, `send()`.
- Как реализовать таймауты? Например, если мы хотим закрывать соединение, в котором какое-то время не было запросов.

Неблокирующий режим файлового дескриптора

```
{
        int sock;
        int flags;

        sock = socket(AF_INET, SOCK_STREAM, 0);

        flags = fcntl(sock, F_GETFL);
        flags |= O_NONBLOCK;
        fcntl(sock, F_SETFL, flags);

        /* Теперь весь ввод-вывод через sock является неблокирующим. */
}

{
        int sock;

        sock = socket(AF_INET, SOCK_STREAM, 0);

        /* Отдельный неблокирующий вызов. */
        send(sock, tx_buf, sizeof(tx_buf), MSG_DONTWAIT);
}
```

Использование неблокирующих вызовов

```
int handle_requests(int data_sock)
{
        while (1) {
                char rx_buf[100];

                ret = recv(data_sock, rx_buf, sizeof(rx_buf), 0);
                if (ret == 0) }
                        break;
                } else if (ret < 0 && errno == EWOULDBLOCK) {
                        printf("No request\n");

                        /* Что делать дальше? */
                } else if (ret < 0) {
                        fprintf(stderr, "Failed to read request\n");
                        return -1;
                }

                /* Обработка запроса. */
        }
}
```

Возможное решение: многопоточность

- Каждое соединение обрабатывается отдельным потоком („нитью”).
- В Linux поток является специальным процессом.
- Все потоки имеют общее адресное пространство с основным процессом.
- Для создания потока используется функция `pthread_create()` вместо `fork()/execv()`,
- Стандартная библиотека предоставляет набор функций для работы с потоками: `pthread_join()`, `pthread_cancel()`, `pthread_attr_init()` и др.
- Потоки исполняются под управлением планировщика процессов Linux.
- Linux автоматически распределяет потоки pthread по имеющимся процессорным ядрам.

Многопоточный сервер

```
int data_socks[MAX_CONNECTIONS];
int next;

void handle_requests(void *data)
{
	int data_sock = (int)data;

	while (1) {
		/* Чтение и обработка запроса. */
	}
}

int handle_connections(int listen_sock)
{
	while (1) {
		pthread_t thread;

		data_socks[next] = accept(listen_sock, NULL, NULL);
		pthread_create(&thread, NULL, handle_requests, (void *)data_socks[next]);
		next++;
	}
}
```

Проблема: конфликт при доступе к общим данным

```
Поток 1          Поток 2          Результат
array[0] = 'a';                   'a', '-'
                 array[0] = 'b';  'b', '-'
                 array[1] = 'b';  'b', 'b'
array[1] = 'a';                   'b', 'a' - недопустимая комбинация
```

Область кода, в которой происходит обращение к общим данным, называется *критической секцией*. Чтобы обеспечить целостность данных, критическая секция должна исполняться не более чем одним потоком в каждый момент времени.

Вход в критическую секцию защищается с помощью мьютексов:

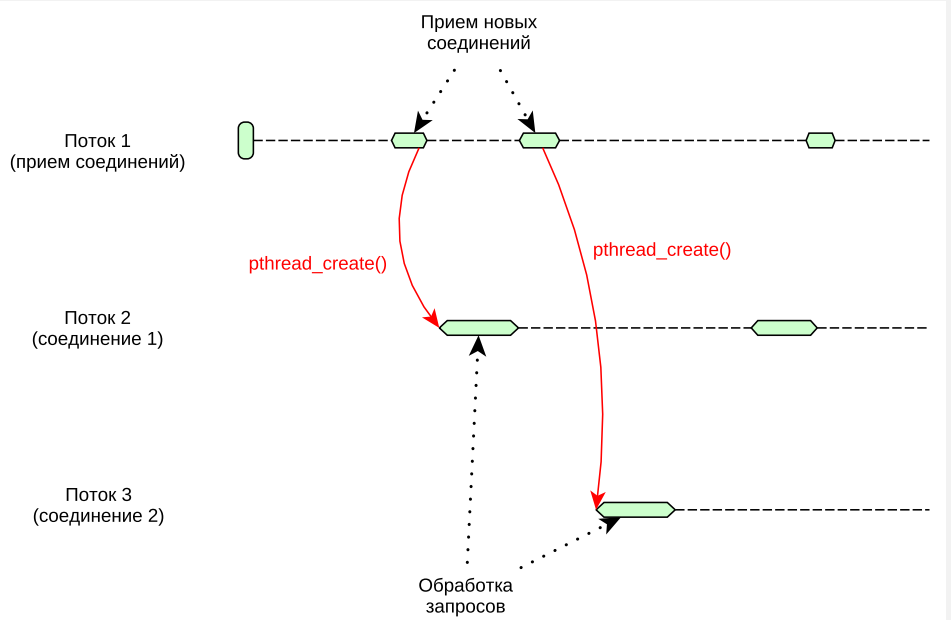
```
mutex_lock();
array[0] = 'a'; array[1] = 'a';
mutex_unlock();
```

Если при входе в критическую секцию поток обнаруживает запертый мьютекс, он ждет, когда другой поток освободит этот мьютекс.

Многопоточный сервер с мьютексами

```
pthread_mutex_t global_mutex;

void handle_requests(void *data)
{
	int data_sock = (int)data;
	while (1) {
		recv(data_sock, rx_buf, sizeof(rx_buf), 0);

		pthread_mutex_lock(&global_mutex);
		/* Обработка запроса. */
		pthread_mutex_unlock(&global_mutex);
	}
}

int handle_connections(int listen_sock)
{
	while (1) {
		pthread_t thread;
		data_socks[next] = accept(listen_sock, NULL, NULL);
		pthread_create(&thread, NULL, handle_requests, (void *)data_socks[next]);
	}
}
```

Нужен ли вообще параллелизм?



parallelism - решение нескольких задач на нескольких ядрах.
concurrency - исполнение нескольких задач поочередно на одном ядре.

Использование гипотетического кооперативного планировщика

```
void handle_requests(void *data)
{
	int data_sock = (int)data;
	while (1) {
		ret = recv(data_sock, rx_buf, sizeof(rx_buf), 0);
		if (ret < 0 && errno == EWOULDBLOCK)
			schedule();

		/* Обработка запроса. */
	}
}

int handle_connections(int listen_sock)
{
	while (1) {
		cooperative_thread_t thread;
		data_socks[next] = accept(listen_sock, NULL, NULL);
		if (data_socks[next] < 0 && errno == EWOULDBLOCK)
			schedule();
		coop_thread_create(&thread, NULL, handle_requests, (void *)data_socks[next]);
	}
}
```

Вызов функций по событиям

```
void event_dispatcher(void)
{
	int listen_sock, data_socks[MAX_CONNECTIONS];

	/* wait_events - гипотетическая функция, которая ждет появления новых
	соединений или данных хотя бы на одном из перечисленных сокетов. */
	wait_events(listen_sock, data_sock[0], ... , data_socks[MAX_CONNECTIONS - 1]);

	/* has_events - гипотетическая функция, которая проверяет наличие новых
	соединений или данных на указанном сокете. */
	if (has_events(listen_sock))
		handle_connections(listen_sock);

	for (i = 0; i < MAX_CONNECTIONS; i++)
		if (has_events(data_socks[i]))
			handle_requests(data_socks[i]);
}
```

Ожидание событий с помощью `select()`

```
int select(int nfds, fd_set *readfds, fd_set *, fd_set *, struct timeval *timeout);
```

- `fd_set` - *множество файловых дескрипторов*.
- Функция `select()` принимает на вход три множества дескрипторов и ждет до тех пор, пока хотя бы один дескриптор в любом множестве не окажется в состоянии *готовности*. Для множества readfds функция ожидает *готовность к чтению*.
- Функция select() меняет множества дескрипторов. После завершения в каждом множестве остаются только дескрипторы в состоянии готовности.
- Готовность к чтению означает, что последующий `read` (или `recv`) вернет данные без ожидания.
- Аргумент `timeout` задает максимальное время ожидания готовности. Если это время истекает, `select` возвращает 0.
- Если `timeout` установлен в `NULL`, `select` ждет готовности неограниченное время.

Работа с множеством файловых дескрипторов `fd_set`

- `fd_set` - битовая маска.
- Задание множества:

```
fd_set fds;
int fd1 = 2, fd2 = 5;
FD_ZERO(&fds);          ...00000000₂
FD_SET(fd1, &fds);      ...00000100₂
FD_SET(fd2, &fds);      ...00100100₂
```

- Проверка принадлежности множеству:

```
int fd1 = 2;
if (FD_ISSET(fd1, &fds)) {
        /* Если принадлежит. */
} else {
        /* Если не принадлежит. */
}
```

- Аргумент `nfds` должен быть равен `(max(fd1, fd2, ..., fdn) + 1)`.

Вызов функций по событиям 2

```
void event_dispatcher(void)
{
	int listen_sock, data_socks[MAX_CONNECTIONS], max_fd;
	fd_set fds;

	FD_ZERO(&fds);
	FD_SET(listen_sock, &fds);
	max_fd = listen_sock;
	for (i = 0; i < MAX_CONNECTIONS; i++) {
		FD_SET(data_socks[i], &fds);
		if (data_socks[i] > max_fd)
			max_fd = data_socks[i];
	}

	select(max_fd + 1, fds, NULL, NULL, NULL);

	if (FD_ISSET(listen_sock, &fds))
		handle_connections(listen_sock);

	for (i = 0; i < MAX_CONNECTIONS; i++)
		if (FD_ISSET(data_socks[i], &fds))
			handle_requests(data_socks[i]);
}
```

Работа со структурой `struct timeval`

- Определение

```
struct timeval {
       time_t      tv_sec;     /* seconds */
       suseconds_t tv_usec;    /* microseconds */
};
```

- Получение текущего времени:

```
struct timeval tv;
gettimeofday(&tv, NULL); /* Время от начала Эпохи (1970 год) */
```

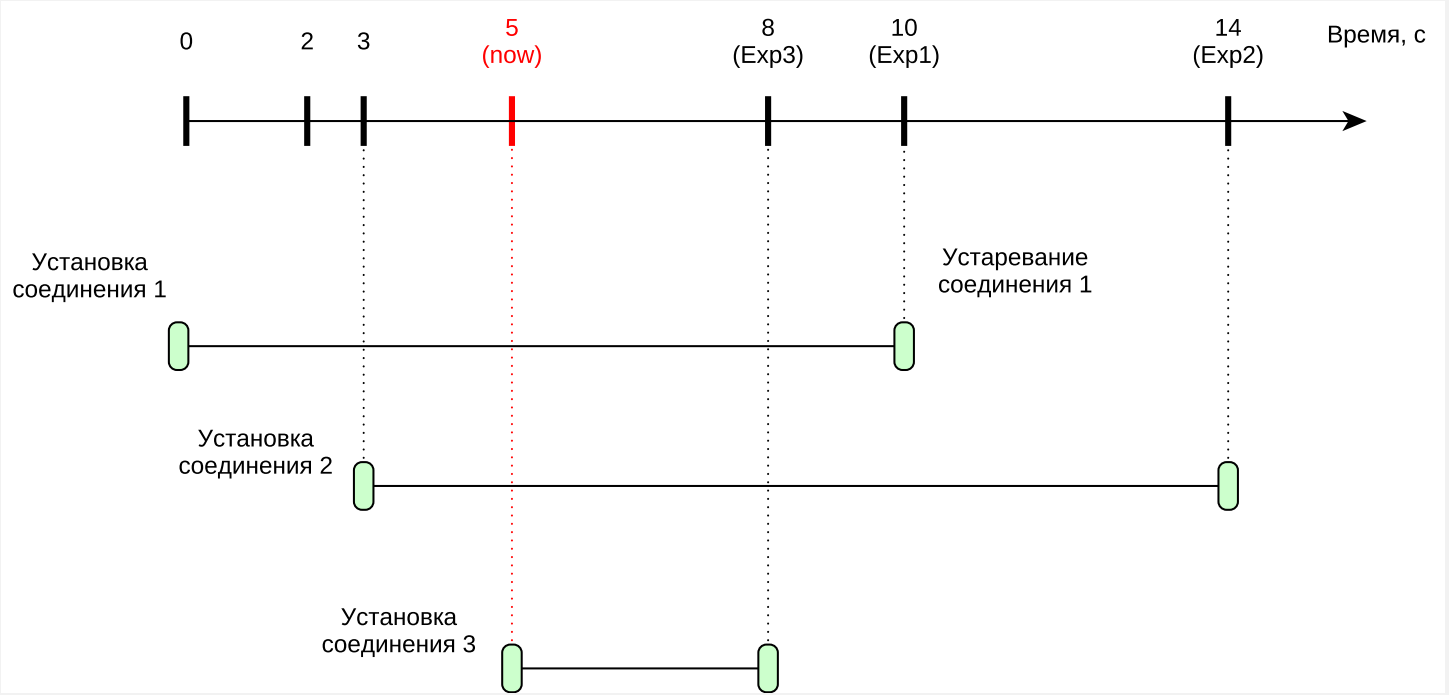
- Арифметика:

```
struct timeval tv1, tv2, newtv;
timeradd(&tv1, &tv2, &newtv); /* newtv = tv1 + tv2 */
timersub(&tv1, &tv2, &newtv); /* newtv = tv1 - tv2 */
if (timercmp(&tv1, &tv2, <) {
       /* tv1 < tv2 */
} else {
       /* tv1 >= tv2 */
}
```

Ожидание запроса с таймаутом

```
/* Время, начиная с которого мы считаем соединение data_socks[0] устаревшим. */
struct timeval connection_expiration_time;

{
        struct timeval now, timeout;

        FD_ZERO(&fds);
        FD_SET(data_socks[0], &fds);
        max_fd = data_socks[0];

        gettimeofday(&now, NULL);
        /* timeout = connection_expiration_time - now */
        timersub(&connection_expiration_time, &now, &timeout);

        ret = select(max_fd + 1, fds, NULL, NULL, &timeout);
        if (ret == 0) {
                /* Таймаут. */
                close(data_socks[0]);
                return 0;
        }
        if (FD_ISSET(data_socks[0], &fds))
                handle_requests(data_socks[0]);
}
```

Расчет таймаута при наличии нескольких соединений



$$timeout = min(Exp_1 - now, Exp_2 - now, ..., Exp_n - now)$$

Работа с несколькими таймерами

```
/* Время, начиная с которого мы считаем соединение data_socks[i] устаревшим. */
struct timeval connection_expiration_times[MAX_CONNECTIONS]

{
	struct timeval now, timeout = {.tv_sec = LONG_MAX}, candidate;

	gettimeofday(&now, NULL);
	for (i = 0; i < MAX_CONNECTIONS; i++) {
		timersub(&connection_expiration_times[i], &now, &candidate);
		if (timercmp(&candidate, &timeout, <)
			timeout = candidate;
	}

	ret = select(max_fd + 1, fds, NULL, NULL, &timeout);
	if (ret == 0) {
		/* Могло пройти больше времени, чем timeout. */
		gettimeofday(&now, NULL);
		for (i = 0; i < MAX_CONNECTIONS; i++) {
			if (timercmp(&connection_expiration_times[i], &now, <)
				close(data_socks[i]);
	}
}
```

Цикл обработки событий (event loop)

```
while (1) {
        struct fd_set fds;
        struct timeval timeout;

        FD_ZERO(&fds);
        for (...) { /* Перебор всех сокетов */
                FD_SET(sock_fd, &fds);
                max_fd = sock_fd > max_fd ? sock_fd : max_fd;

                ... /* Модификация timeout */
        }
        ret = select(max_fd + 1, &fds, NULL, NULL, &timeout);
        if (ret == 0) {
                /* Обработка таймаута. */
        } else {
                for (...) { /* Перебор всех сокетов */
                        if (FD_ISSET(sock_fd, &fds)) {
                                /* Обработка нового соединения или запроса. */
                        }
                }
        }
}
```

Правила написания обработчиков событий

- Не должно быть блокирующих вызовов.
- Не должно быть больше одного неблокирующего вызова, который может вернуть ошибку `EWOULDBLOCK`.

Так делать нельзя:

```
int handle_requests(int data_sock)
{
        recv(data_sock, request1, sizeof(request1), MSG_DONTWAIT);
        /* Обработка запроса 1. */
        send(data_sock, response1, sizeof(response1), 0);

        recv(data_sock, request2, sizeof(request2), MSG_DONTWAIT);
        /* Обработка запроса 2. */
        send(data_sock, response2, sizeof(response2), 0);

        return 0;
}
```

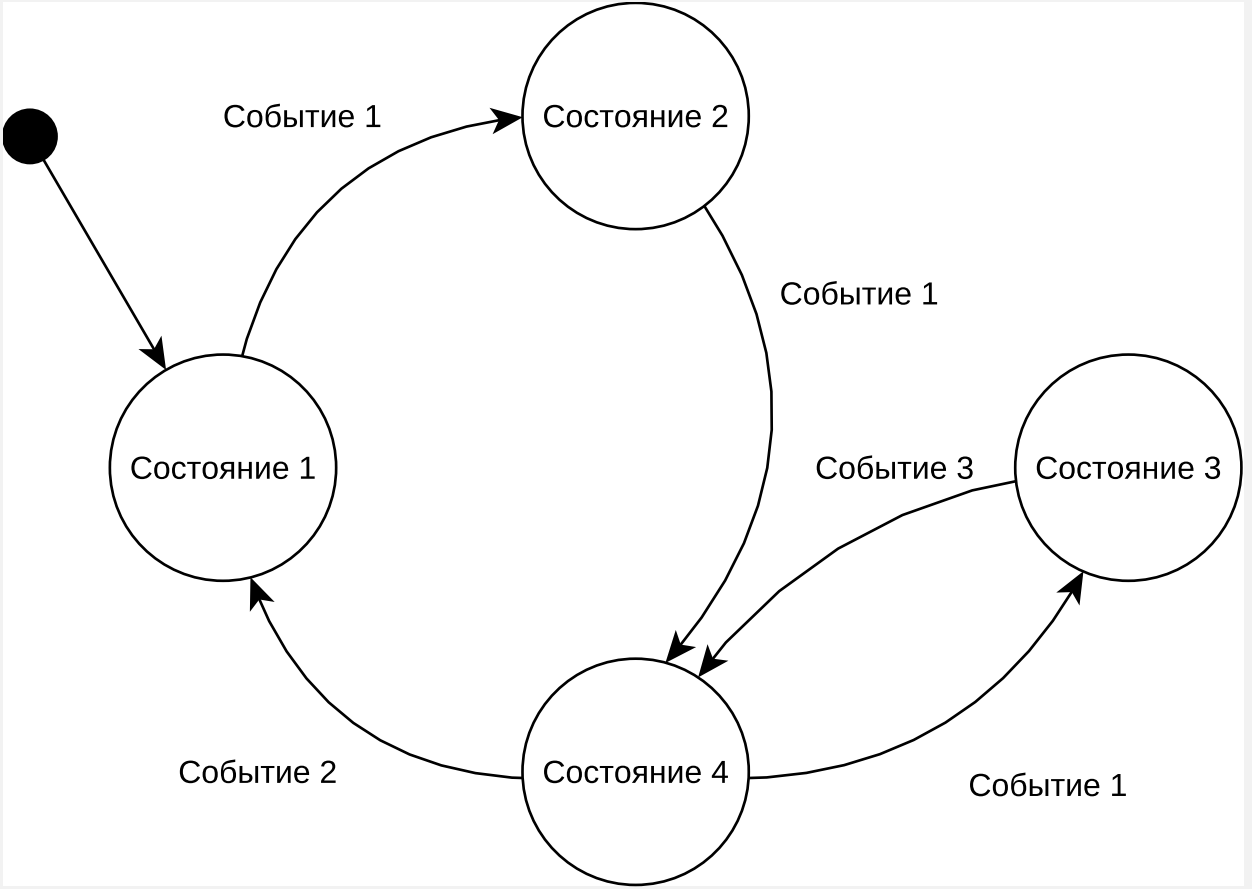
Правила написания обработчиков событий (2)

```
int num_requests;

int handle_requests(int data_sock)
{
       switch (num_requests) {
       case 0:
              recv(data_sock, request1, sizeof(request1), MSG_DONTWAIT);
              /* Обработка запроса 1. */

              /* Предполагаем, что send() никогда не блокирует программу. */
              send(data_sock, response1, sizeof(response1), 0);
              nrequests++;
              break;
       case 1:
              recv(data_sock, request2, sizeof(request2), MSG_DONTWAIT);
              /* Обработка запроса 2. */
              send(data_sock, response2, sizeof(response2), 0);
              nrequests++;
              break;
       }
       return 0;
}
```

Больше, чем просто `select()`

- Помимо `select()`, Linux предлагает функционально эквивалентные API `poll()` и `epoll()`.
- Функция `select()` реализует шаблон (паттерн) проектирования „реактор”.
- Библиотеки libevent и libev для разработки высокопроизводительных серверов основаны на вызовах `select()`/`poll()`.
- В большинстве языков программирования есть библиотеки, предоставляющие аналогичные функции: Twisted в Python, EventMachine в Ruby и др.
- В некоторых языках программирования (JavaScript) весь ввод-вывод осуществляется через планирование и обработку событий.

Диаграмма состояний



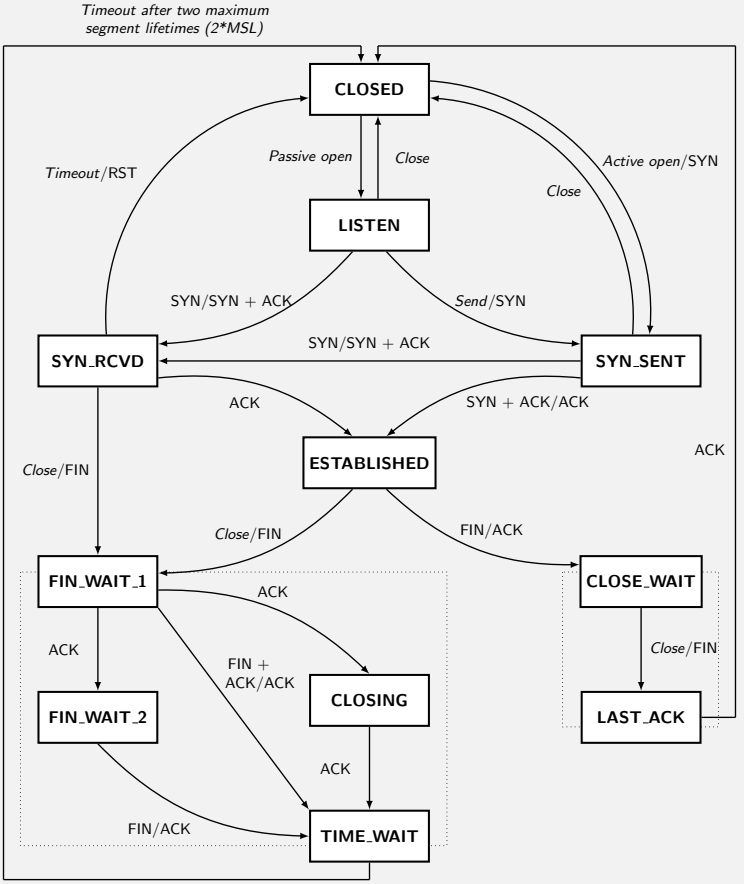
Конечный автомат

Конечный автомат Мура: $(\Sigma, S, s_0, \delta, F)$

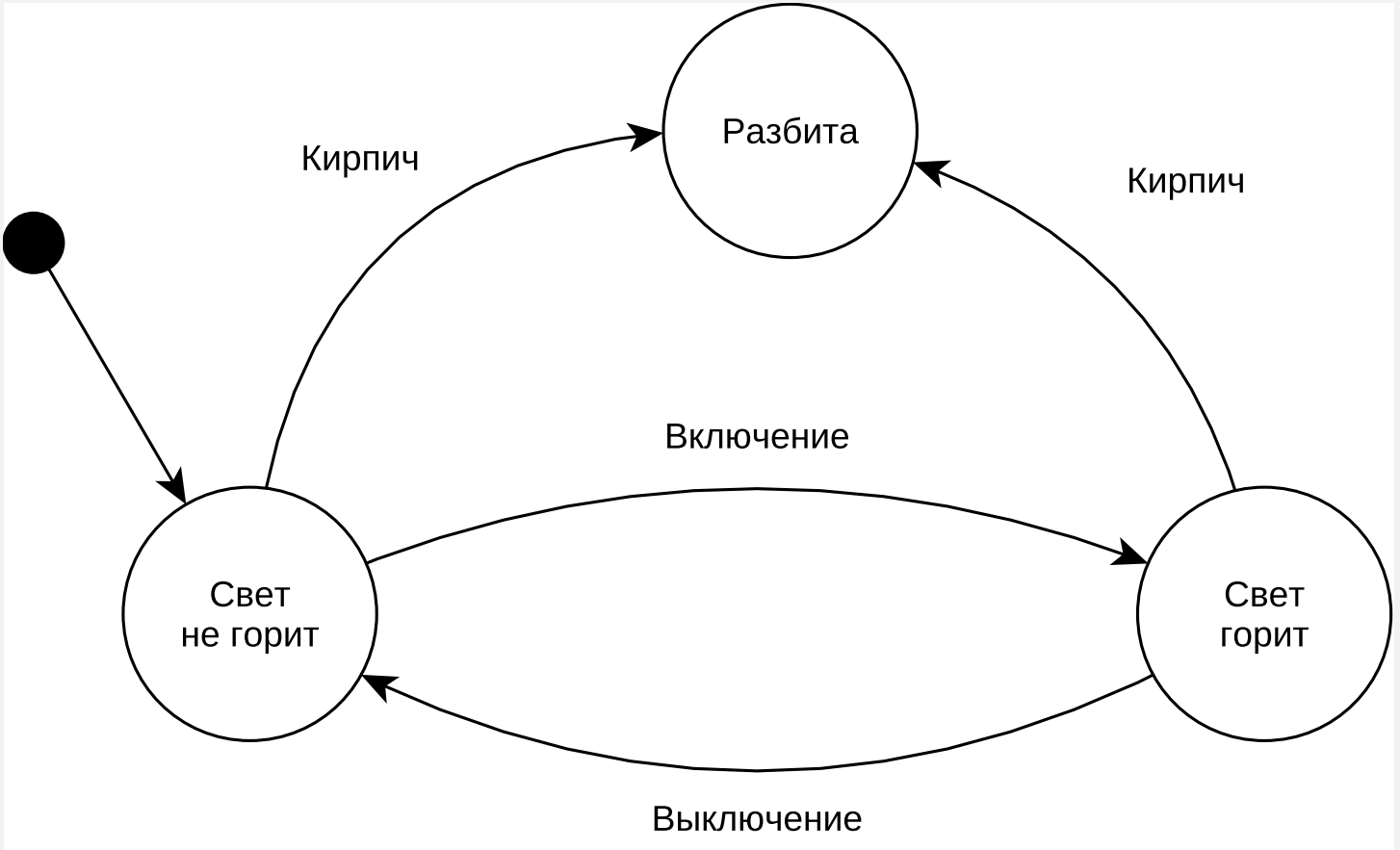
- Σ - множество входных событий.
- S - множество состояний.
- s_0 - начальное состояние.
- δ - функция переходов, $\delta : S \times \Sigma \to S$.
- F - множество конечных состояний.

Автомат Мили: для каждого перехода между состояниями дополнительно задается выходной сигнал. На диаграмме состояний событие и выходной сигнал разделяются символом ’/’.

Конечный автомат TCP соединения



Пример: лампочка



Реализация (1)

```
enum state {
       LIGHT_ON,
       LIGHT_OFF,
       BROKEN,
};

enum events {
       TURN_ON,
       TURN_OFF,
       BRICK,
};

static enum state state;
```

Реализация (2)

```
void run_state_machine(enum event event)
{
	switch (state) {
	case LIGHT_OFF:
		if (event == TURN_ON)
			state = LIGHT_ON;
		else if (event == BRICK)
			state = BROKEN;
		break;
	case LIGHT_ON:
		if (event == TURN_OFF)
			state = LIGHT_OFF;
		else if (event == BRICK)
			state = BROKEN;
		break;
	case BROKEN:
		break;
	}
}
```

Реализация (3)

```
int main(int argc, char *argv[])
{
	enum event event;

	while (1) {
		char command[128];

		fgets(command, sizeof(command), stdin);
		if (!strncmp(command, "turn-on", 7))
			event = TURN_ON;
		else if (!strncmp(command, "turn-off", 8))
			event = TURN_OFF;
		else if (!strncmp(command, "brick", 5))
			event = BRICK;

		run_state_machine(event);
	}

	return 0;
}
```

Конечный автомат + select()

- Поскольку переход между состояниями меняет внутреннее состояние программы и обычно не требует блокирующих вызовов, функция перехода может безопасно вызываться из цикла обработки событий. Это позволяет поочередно исполнять несколько экземпляров одного конечного автомата в одном многозадачном приложении. Каждый экземпляр в нем можно рассматривать как одну задачу.

- Программа должна выполнять трансляцию низкоуровневых событий, детектируемых с помощью функции select(), в высокоуровневые события, подаваемые на вход конечного автомата. Примеры низкоуровневых событий: чтение запроса из сокета, истечение таймаута, разрыв соединения, установка нового соединения. Высокоуровневые события представляются переменными перечислимого типа.

Многозадачный сервер

- Все входные сигналы (новые соединения, новые запросы, таймауты) обрабатываются циклом обработки событий на основе вызова `select()`.
- Логика обслуживания каждого отдельного соединения описывается конечным автоматом.
- При установке нового соединения программа создает новый экземпляр конечного автомата.
- При получении нового запроса или при срабатывании таймаута программа определяет экземпляр конечного автомата (соединение), к которому они относятся, и передает их на вход автомата в качестве событий.
- При переходе между состояниями автомата программа меняет внутреннее состояние (задает переменные, устанавливает таймеры), а также формирует выходные сигналы (ответы на запросы и т.д.).

Задача: повторитель символов