Семинар 20: мультиплексирование I/O-операций

13 мая, 2020

Что делать, если нужно обработать несколько клиентов одновременно?

Apache MPM

- MPM = multiprocessing module
- Модуль Apache, который распараллеливает обработку запросов
- Три режима работы: prefork/worker и event

Apache MPM: prefork model

- Для каждого соединения стартует новый процесс
- Количество inflight-соединений = количество процессов
- Поддерживает какое-то количество незанятых (spare) процессов — поэтому *pre*-fork
- Обеспечивал безопасность, однако тратил много ресурсов

Apache MPM: prefork model

```
for (;;) {
    int client_fd = accept(...);

    while (children.size() > MaxRequestsNum) {
        pid_t finished = wait(NULL);
        children.erase(finished);
    }

    pid_t pid = fork();
    if (pid == 0) {
        process_connection(client_fd);
    }

    children.push_back(pid);
}
```

Apache MPM: worker model

- Как сократить количество ресурсов?
- Будем использовать треды вместо процессов!
- Несколько процессов-контейнеров, чтобы один тред не мог сломать весь веб-сервер

HTTP keep alive

- На каждый HTTP-запрос устанавливается отдельное TCP-соединение
- То есть на каждый запрос тратится дополнительно $\frac{3}{2}RTT$ на 3WH
- Зачем закрывать соединение? Давайте делать следующий запрос в том же!
- С HTTP/1.1 все соединения по-умолчанию считаются keep alive
- prefork/worker model + keep alive = :(
- Очень много idle-клиентов тратят слишком много ресурсов
- Можно устроить DoS почти не тратя CPU

Apache MPM: event model

- Основан на worker
- Пусть будет один тред следит за всеми соединениями
- Воркеры обрабатывают только уже отдельные запросы
- Проблема: как в одном треде уследить за несколькими соединениями?

Busy waiting

```
for (;;) {
    for (int conn_fd : connections) {
        char unused;
        ssize_t res = recv(conn_fd, &unused, 1, MSG_PEEK);
        if (res > 0) {
            // что-то появилось в conn_fd,
            // можно отдавать в воркера
        } else if (errno == EAGAIN) {
            // пока ничего нет :(
        } else {
            // res == 0 или ошибка (но не EAGAIN),
            // можно закрыть соединение
            close(conn_fd);
        }
    }
}
```

I/O multiplexing

- `select`, `poll`, `epoll`
- Различаются в деталях, но суть одна – они ждут пока в *хотя бы в одном* из файловых дескрипторов произойдёт какое-либо событие
- Появились данные для чтения – `POLLIN`
- Появились данные для записи – `POLLOUT`
- Произошла ошибка (например, `EPIPE`) – `POLLERR`

I/O multiplexing: poll

```c
#include <poll.h>

struct pollfd {
    int   fd;          /* file descriptor */
    short events;      /* requested events */
    short revents;     /* returned events */
};

int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

I/O multiplexing: select

```c
#include <sys/select.h>

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

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

I/O multiplexing: poll & select

- Самый старые интерфейсы: `poll` появился ещё в прошлом тысячелетии (2.1.x), а `select` в 00-ых (2.6.x)
- Почти все POSIX-совместимые ОС реализуют select, самый переносимый вариант
- `fd_set` ограничен 128 байтами, $\Rightarrow$ нельзя использовать для файловых дескрипторов больше 1024
- Если файловых дескрипторов слишком много, то происходит много копирований (как в userspace, так и при вызове `poll`)

I/O multiplexing: epoll

```c
#include <sys/epoll.h>

struct epoll_event {
    uint32_t     events;
    union {
        void    *ptr;
        int      fd;
        uint32_t u32;
        uint64_t u64;
    } data;
};

int epoll_create1(int flags);

int epoll_ctl(int epfd, int op, int fd,
                        struct epoll_event *event);

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

I/O multiplexing: epoll

- Немного меняется семантика: в epoll нужно регистрировать (`EPOLL_CTL_ADD`) файловые дескрипторы, а затем `epoll_wait` возвращает события, которые произошли с ними
- С каждым файловым дескриптором можно ассоциировать какие-то данные (например, указатель на класс, которые отвечает за соединение)
- Файловые дескрипторы можно подписывать на разные события: `EPOLLIN`, `EPOLLOUT`, `EPOLLHUP`, ...
- Под файловыми дескрипторами могут скрываться сокеты, пайпы, а также специальные механизмы, вроде `eventfd` или `timerfd`
- **ВАЖНО**: epoll на регулярных файлах всегда будет сразу выдавать готовые события

epoll: edge-triggered vs level-triggered

- Разные способы оповещения о событиях для конкретных файловых дескрипторов
- Level-triggered: оповещения будут приходить пока вы полностью не исчерпаете событие (например, не прочитаете весь read buffer)
- Edge-triggered: оповещение будет приходить **только один раз**, как только вы его получите через `epoll_wait`, epoll о нём забудет

Coroutines

- Почему бы внутри программы не попробовать самим управлять потоками?
- Корутины (или green threads, или greenlets) – треды, которые управляются программой
- Обычно используется cooperative multitasking
- Корутины работают, пока не дойдут до блокирующей I/O-операции, затем передают управление другим
- Как только running корутин не осталось, корутиновый движок зависает на epoll/poll/select
- Затем пробуждаются корутины и итерация повторяется

Спасибо!