

## Содержание

<b>1 Введение: Эволюция сетевых архитектур</b>	<b>1</b>
<b>2 Многопоточная модель (Thread-per-Connection)</b>	<b>2</b>
2.1 Ограничения многопоточности . . . . .	2
<b>3 Ограничения ОС: Файловые дескрипторы и ulimit</b>	<b>2</b>
<b>4 Событийная модель: Мультиплексирование через ePoll</b>	<b>2</b>
4.1 Флаг O_NONBLOCK . . . . .	2
4.2 Механизм ePoll . . . . .	3
<b>5 Сравнительный анализ производительности</b>	<b>3</b>
<b>6 Visuals: Сравнение архитектур</b>	<b>3</b>
<b>7 Унифицированный цикл событий: TimerFD, PIDFD и io_uring</b>	<b>4</b>
<b>8 Таймеры как дескрипторы: TimerFD</b>	<b>4</b>
8.1 Механика настройки . . . . .	5
<b>9 События процессов: PIDFD</b>	<b>5</b>
<b>10 Путь к Zero Syscall I/O: io_uring</b>	<b>5</b>
10.1 Идея батчинга (Batching) . . . . .	5
<b>11 Визуализация: Унифицированный Event Loop</b>	<b>6</b>
<b>12 Задача с семинара: Обработка сигналов через FD</b>	<b>6</b>
<b>13 Механика сигналов: Аппаратные корни и безопасность стека</b>	<b>7</b>
13.1 Генезис сигналов: от аппаратных прерываний к программным . . . . .	7
13.2 Анатомия доставки сигнала и манипуляция стеком . . . . .	7
13.3 x86_64 Red Zone и листовые функции . . . . .	7
13.4 Проблема Signal-Safety: почему printf — это риск . . . . .	8
13.5 Безопасные системные вызовы . . . . .	8
<b>14 Синхронная обработка и атомарное ожидание: sigsuspend</b>	<b>9</b>
<b>15 Проблемы пассивного и активного ожидания</b>	<b>9</b>
15.1 Оптимизации компилятора и volatile . . . . .	9
<b>16 Манипуляция масками сигналов</b>	<b>10</b>
<b>17 Критическая гонка (Race Condition): pause()</b>	<b>10</b>

18 Решение: системный вызов sigsuspend	10
19 Итоги раздела	11

## 1 Введение: Эволюция сетевых архитектур

Разработка высокопроизводительных сетевых сервисов требует глубокого понимания механизмов взаимодействия между пользовательским пространством (User Space) и ядром ОС (Kernel Space). Основной метрикой эффективности веб-сервера является не только пропускная способность, но и способность масштабироваться при росте количества одновременных соединений без деградации времени отклика.

## 2 Многопоточная модель (Thread-per-Connection)

Классическая архитектура сетевого сервера базируется на создании отдельного потока выполнения для каждого входящего соединения. Процесс обработки в этом случае линейен: поток вызывает блокирующий системный вызов `accept()`, получает дескриптор соединения и переходит к чтению/записи данных.

### Определение: Context Switch (Переключение контекста)

Процедура сохранения состояния текущего потока (регистры, указатель стека, программный счетчик) и восстановления состояния другого потока. В Linux переключение контекста управляет планировщиком задач и требует перехода в режим ядра, что сопряжено с накладными расходами на кэш-промахи и сброс конвейера процессора.

### 2.1 Ограничения многопоточности

При малом количестве соединений (десятки) данная модель эффективна благодаря простоте программирования. Однако при увеличении числа клиентов до тысяч возникают следующие проблемы:

- Расход памяти:** Каждый поток требует собственного стека (обычно от 2 до 8 МБ в зависимости от настроек `ulimit`).
- Деградация планировщика:** Постоянные переключения между тысячами активных потоков приводят к тому, что значительная часть ресурсов CPU тратится на обслуживание инфраструктуры ОС, а не на полезную нагрузку.

## 3 Ограничения ОС: Файловые дескрипторы и ulimit

Для обработки большого количества соединений необходимо учитывать системные лимиты на количество открытых файловых дескрипторов ([File Descriptor \(FD\)](#)).

### Примечание

По умолчанию в большинстве дистрибутивов Linux лимит на количество открытых файлов процессом составляет 1024. При попытке открыть 2000 соединений сервер вернет ошибку `EMFILE` (Too many open files).

Для изменения лимитов используется команда `ulimit -n` или редактирование конфигурации `/etc/security/limits.conf`. Серверные приложения должны уметь обрабатывать это ограничение, увеличивая лимит через системный вызов `setrlimit()`. Каждое

сетевое соединение — это запись в системной таблице открытых файлов, привязанная к Process Control Block (PCB) (в Linux — `task_struct`).

## 4 Событийная модель: Мультиплексирование через ePoll

Альтернативой многопоточности является мультиплексирование ввода-вывода. Вместо того чтобы блокировать поток на чтении из одного сокета, мы заставляем один поток следить за множеством сокетов одновременно.

### 4.1 Флаг O\_NONBLOCK

Ключевым элементом событийной модели является перевод файловых дескрипторов в неблокирующий режим.

```
1 int flags = fcntl(fd, F_GETFL, 0); // Get current flags
2 fcntl(fd, F_SETFL, flags | O_NONBLOCK); // Set non-blocking flag
```

Листинг 1 – Setting socket to non-blocking mode

В этом режиме системный вызов `read()` или `write()`, если данные недоступны, немедленно возвращает `-1` и устанавливает `errno` в `EAGAIN` или `EWROULDBLOCK`.

### 4.2 Механизм ePoll

`event poll (ePoll)` — это масштабируемый интерфейс уведомления о событиях ввода-вывода в Linux. Он эффективнее устаревших `select` и `poll`, так как сложность уведомления составляет  $O(1)$ , а не  $O(N)$ .

```
1 int epfd = epoll_create1(0); // Create epoll instance
2 struct epoll_event event, events[MAX_EVENTS];
3
4 // Add server socket to ePoll
5 event.events = EPOLLIN;
6 event.data.fd = server_fd;
7 epoll_ctl(epfd, EPOLL_CTL_ADD, server_fd, &event);
8
9 while (1) {
10     // Wait for events (timeout = -1 means infinite)
11     int n = epoll_wait(epfd, events, MAX_EVENTS, -1);
12     for (int i = 0; i < n; i++) {
13         if (events[i].data.fd == server_fd) {
14             // Logic: accept() new connections
15         } else {
16             // Logic: non-blocking read/write for clients
17         }
18     }
19 }
```

Листинг 2 – Main multiplexing loop using ePoll

#### Определение: Инверсия управления (State Machine)

При использовании `ePoll` программист не контролирует порядок исполнения логики линейно. Код превращается в конечный автомат (State Machine), где обработка данных разбивается на части, привязанные к готовности дескриптора. Это требует сохранения состояния (Context) сессии вручную между вызовами событий.

## 5 Сравнительный анализ производительности

На семинаре было проведено тестирование двух реализаций сервера: на базе потоков и на базе `ePoll`. Нагрузка создавалась Python-скриптом, имитирующим 2000 одновременных соединений с передачей 1 байта данных.

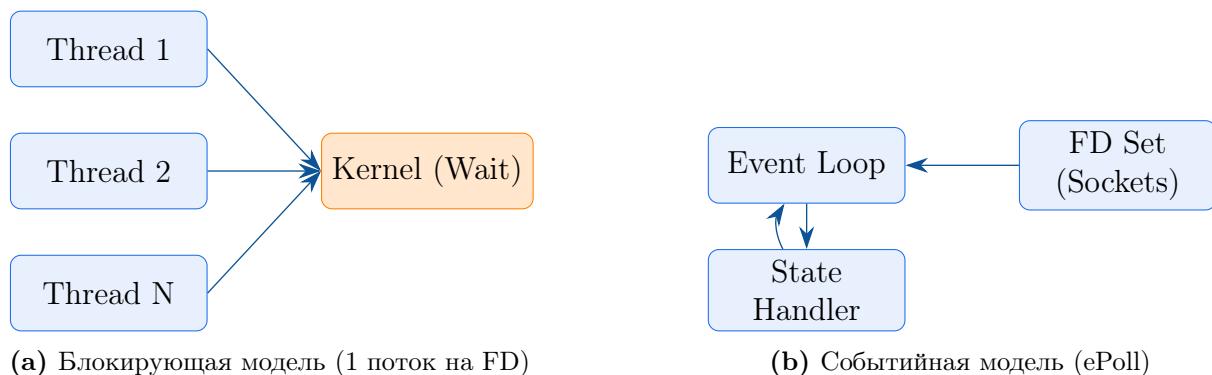
**Таблица 1** – Метрики производительности при 2000 соединениях

Архитектура	Потребление CPU	Масштабируемость
Multi-threaded	35–40%	Низкая (лимит потоков)
<code>ePoll</code> Reactor	~25%	Высокая ( $O(1)$ на событие)

Разница в 10–15% потребления CPU объясняется отсутствием избыточных переключений контекста и эффективным использованием кэша L1 в одном потоке исполнения.

## 6 Visuals: Сравнение архитектур

На рис. 1 показано различие в обработке запросов между блокирующими и событийной моделями.



**Рис. 1** – Сравнение архитектур сетевых серверов

### Итоги раздела

- **Многопоточность** интуитивно понятна, но не масштабируется из-за накладных расходов на Context Switch и потребление RAM стеками.
- **Мультиплексирование (ePoll)** позволяет эффективно использовать один поток CPU для обслуживания тысяч соединений.
- **Неблокирующий I/O** и флаг `O_NONBLOCK` являются обязательным условием работы событийного цикла.
- **Цена ePoll** – значительное усложнение кода и необходимость реализации логики сервера в виде конечного автомата.

## 7 Унифицированный цикл событий: TimerFD, PIDFD и io\_uring

Развитие механизмов мультиплексирования в Linux привело к концепции «единого дескриптора», где не только сетевые сокеты, но и таймеры, сигналы и события жизненного цикла процессов представляются в виде файловых дескрипторов. Это позволяет строить архитектуры, в которых весь ввод-вывод и управляющая логика сосредоточены в одном вызове `epoll_wait`.

## 8 Таймеры как дескрипторы: TimerFD

Традиционные методы работы с временем в системном программировании, такие как `nanosleep()` или `setitimer()`, имеют существенный недостаток: они либо блокируют поток, либо требуют асинхронной обработки через сигналы, что нарушает событийную логику `ePoll`.

### Определение: TimerFD

Механизм ядра Linux, создающий файловый дескриптор, который становится доступным для чтения (`EPOLLIN`) по истечении заданного интервала времени. Это позволяет интегрировать временные события непосредственно в цикл мультиплексирования.

### 8.1 Механика настройки

Таймер описывается структурой `itimerspec`, состоящей из двух величин: `it_value` (время до первого срабатывания) и `it_interval` (период последующих срабатываний).

```

1 struct itimerspec new\_value;
2 new\_value.it\_value.tv\_sec = 1; // First expiration after 1 second
3 new\_value.it\_value.tv\_nsec = 0;
4 new\_value.it\_interval.tv\_sec = 2; // Repeat interval: 2 seconds
5 new\_value.it\_interval.tv\_nsec = 0;
6
7 // Create non-blocking timer descriptor
8 int tfd = timerfd\_create(CLOCK\_MONOTONIC, TFD\_NONBLOCK);
9 // Arm the timer
10 timerfd\_settime(tfd, 0, &new\_value, NULL);

```

Листинг 3 – Periodic timer initialization using `timerfd_settime`

### Примечание

При срабатывании таймера дескриптор возвращает результат при вызове `read()`. Возвращаемое значение – это 8-байтовое беззнаковое целое число (`uint64_t`), представляющее количество произошедших срабатываний с момента последнего чтения. Это критически важно для обнаружения «пропусков» (overruns), если основной поток был занят другой работой.

## 9 События процессов: PIDFD

Долгое время интеграция завершения дочерних процессов в событийные циклы была затруднена. Сигнал `SIGCHLD` асинхронен, а вызовы семейства `wait()` блокируют поток. Относительно недавняя абстракция `pidfd` (доступна с ядер 5.2+) решает эту проблему.

### Определение: PIDFD

Файловый дескриптор, ссылающийся на конкретный процесс. Он становится «готовым к чтению», когда соответствующий процесс завершается.

Использование `pidfd` вместо традиционных PID предотвращает состояние гонки (Race Condition), когда PID завершенного процесса переиспользуется операционной системой для нового процесса до того, как родитель успел вызвать `waitid()`. В контексте `ePoll` это позволяет обрабатывать завершение дочерних задач так же, как чтение из сокета.

## 10 Путь к Zero Syscall I/O: io\_uring

Несмотря на эффективность `ePoll`, он все еще требует минимум одного системного вызова (`epoll_wait`) для получения событий и последующих вызовов (`read`, `write`) для обработки данных. Для высоконагруженных систем это создает оверхед на переключение между User и Kernel mode.

### 10.1 Идея батчинга (Batching)

Современное решение — `io_uring`. Оно базируется на разделяемой памяти между ядром и приложением в виде двух кольцевых буферов:

1. **Submission Queue (SQ):** Приложение записывает сюда запросы на ввод-вывод.
2. **Completion Queue (CQ):** Ядро записывает сюда результаты выполнения.

#### Примечание

В предельном режиме (`IORING_SETUP_SQPOLL`) ядро выделяет отдельный ядерный поток, который сам сканирует SQ. Приложение просто кладет данные в память и забирает результаты без единого системного вызова в основном цикле.

## 11 Визуализация: Унифицированный Event Loop

На рис. 2 представлена архитектура современного сетевого рантайма, объединяющего разнородные ресурсы.

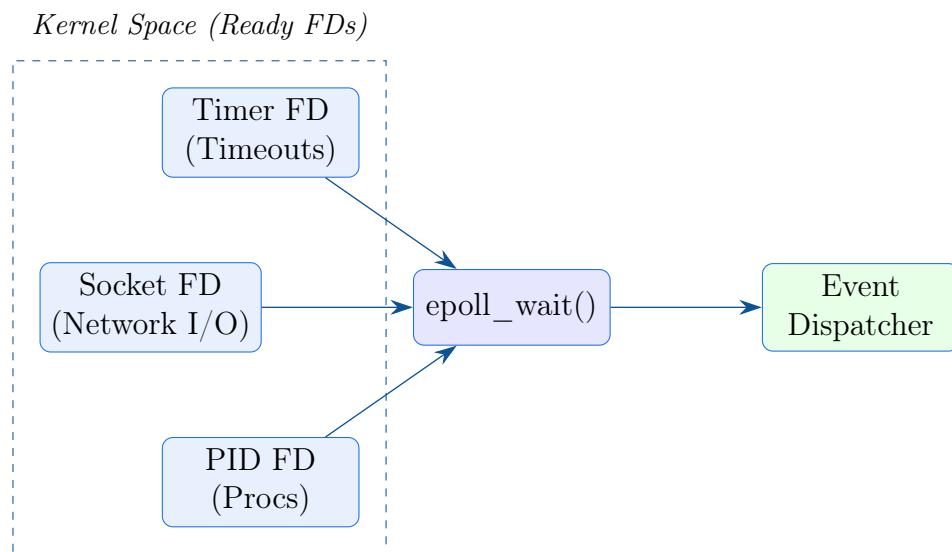


Рис. 2 – Схема унификации ресурсов в событийном цикле

## 12 Задача с семинара: Обработка сигналов через FD

На семинаре обсуждалась возможность интеграции даже классических сигналов в этот цикл через `signalfd()`. Это позволяет избежать написания небезопасных (signal-unsafe) обработчиков, перенося логику обработки сигнала в обычный синхронный поток.

## Итоги раздела

- **TimerFD** позволяет обрабатывать таймауты без прерывания логики цикла и без накладных расходов на сигналы.
- **PIDFD** решает проблему зомби-процессов и Race Condition при мониторинге дочерних задач.
- **Batching** и **io\_uring** представляют собой вершину эволюции I/O в Linux, стремясь исключить системные вызовы из «горячего цикла» обработки.
- Единый цикл событий упрощает архитектуру, превращая все внешние воздействия в последовательность дескрипторов.

## 13 Механика сигналов: Аппаратные корни и безопасность стека

Концепция сигналов в операционных системах семейства POSIX является прямой программной надстройкой над механизмом аппаратных прерываний (Interrupts) и исключений (Exceptions) процессора. Понимание этой связи критично для написания корректного системного кода, так как обработка сигнала нарушает линейную логику исполнения программы и вводит скрытый параллелизм внутри одного потока.

### 13.1 Генезис сигналов: от аппаратных прерываний к программным

На аппаратном уровне процессор реагирует на внешние события (I/O) или ошибки исполнения (деление на ноль, неверный адрес) через таблицу дескрипторов прерываний (IDT). Операционная система перехватывает эти события и транслирует их в абстракцию сигналов для пользовательских процессов.

#### Определение: Сигнал

Асинхронное уведомление процесса о событии. Сигналы могут генерироваться аппаратно (например, SIGSEGV при обращении к незамапленной странице) или программно через системный вызов `kill()`.

При возникновении исключения, такого как **Page Fault**, управление переходит в ядро. Ядро анализирует причину и, если ошибка произошла в User Mode, выставляет соответствующий бит в маске ожидающих сигналов (*pending signals*) в **PCB** процесса.

### 13.2 Анатомия доставки сигнала и манипуляция стеком

Когда ядро планирует возврат процесса из режима ядра в пользовательский режим, оно проверяет наличие необработанных сигналов. Если для сигнала установлен пользовательский обработчик (*handler*), ядро выполняет процедуру «инъекции» вызова:

- Сохранение контекста:** Состояние регистров (RAX, RIP, EFLAGS и др.) сохраняется в специальную структуру на стеке пользователя — *Signal Frame*.
- Манипуляция RIP/RSP:** Ядро принудительно изменяет указатель команд (RIP) на адрес обработчика сигнала и корректирует указатель стека (RSP).
- Трамплин (Restorer):** На стек также кладется адрес кода «возврата» (`sigreturn`), который вызовет системный вызов для восстановления исходного контекста после завершения обработчика.

### 13.3 x86\_64 Red Zone и листовые функции

Важнейшим аспектом безопасности стека в архитектуре x86\_64 является понятие «красной зоны» (Red Zone). Согласно ABI (Application Binary Interface), область в 128 байт ниже текущего значения RSP считается зарезервированной.

#### Примечание

Листовые функции (те, что не вызывают другие функции) могут использовать Red Zone для хранения локальных переменных без явного изменения RSP. Это оптимизация, позволяющая экономить такты процессора на манипуляциях со стеком.

Чтобы не повредить данные в Red Zone, ядро при создании Signal Frame обязано сместить указатель стека еще на 128 байт глубже. Если бы ядро этого не делало, обработчик сигнала затер бы локальные переменные прерванной функции.

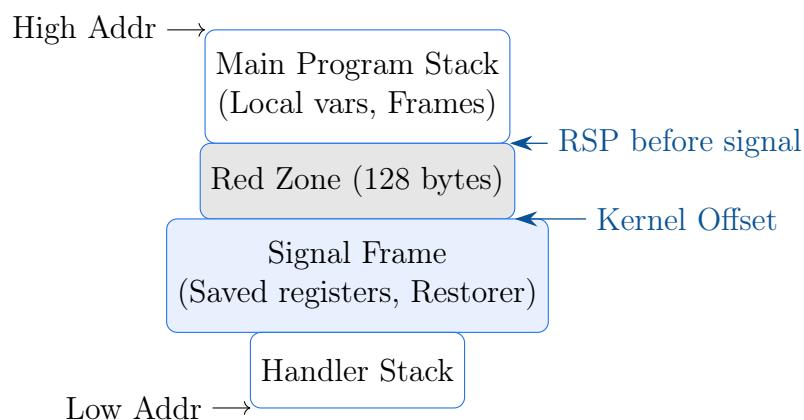


Рис. 3 – Stack layout during signal processing on x86\_64

### 13.4 Проблема Signal-Safety: почему printf — это риск

Поскольку сигнал может прервать программу в произвольной точке (между любыми двумя инструкциями ассемблера), возникает проблема реентерабельности (Reentrancy). Большинство функций стандартной библиотеки C (libc) не являются *Async-Signal-Safe*.

```

1 void handler(int sig) {
2     // DANGEROUS: printf takes an internal mutex for the output stream
3     printf("Received signal %d\n", sig);
4 }
5
6 int main() {
7     signal(SIGINT, handler);
8     while(1) {
9         // If the signal arrives while printf already holds the mutex,
10        // a Deadlock occurs: the handler will wait for the same mutex forever.
11         printf("Working...\n");
12     }
13 }
```

Листинг 4 – Example of a dangerous handler (Signal-Unsafe)

Аналогичная проблема касается `malloc()` и `free()`: они управляют глобальными структурами данных кучи под блокировками. Прерывание процесса во время модифика-

ции связного списка блоков памяти приведет к повреждению кучи (*Heap Corruption*) или вечной блокировке.

### 13.5 Безопасные системные вызовы

Для корректной работы внутри обработчика можно использовать только ограниченный набор функций, определенных стандартом POSIX как атомарные относительно сигналов.

**Таблица 2 – Examples of Async-Signal-Safe functions**

Function	Description
<code>write()</code>	Direct write to descriptor (no libc buffering)
<code>read()</code>	Read from descriptor
<code>_exit()</code>	Immediate termination without calling <code>atexit</code> functions
<code>kill()</code>	Sending a signal
<code>signal()</code>	Setting a signal handler

### Итоги раздела

- Сигналы — это программные прерывания, управляемые ядром через манипуляцию RIP и стеком пользователя.
- Red Zone** (128 байт) защищает данные листовых функций от затирания обработчиками сигналов на архитектуре x86\_64.
- Основная угроза в обработчиках — **Deadlock** из-за неатомарных функций libc (`printf`, `malloc`).
- Золотое правило: обработчик должен быть максимально простым, в идеале — только выставлять флаг типа `volatile sig_atomic_t`.

## 14 Синхронная обработка и атомарное ожидание: `sigsuspend`

Асинхронная природа сигналов накладывает жесткие ограничения на используемые функции. Для обхода проблем реентерабельности и неопределенного состояния памяти в системном программировании применяется паттерн синхронного ожидания: вместо выполнения сложной логики в обработчике, программа переходит в состояние сна до момента доставки сигнала, который лишь выставляет флаг готовности.

## 15 Проблемы пассивного и активного ожидания

Простейший способ дождаться сигнала — использование глобального флага. Однако реализация данного подхода сталкивается с двумя типами проблем: архитектурными и компиляторными.

```

1 int flag = 0;
2 void handler(int sig) { flag = 1; }
3
4 int main() {
5     signal(SIGINT, handler);
6     while (!flag); // Busy wait: 100% CPU load
7     return 0;
8 }
```

**Листинг 5 – Busy Wait: incorrect implementation**

### 15.1 Оптимизации компилятора и volatile

В приведенном примере компилятор при высоком уровне оптимизации (например, -O3) может предположить, что переменная `flag` не меняется внутри цикла, так как в теле цикла нет обращений к ней. В результате проверка выносится за пределы цикла, и программа входит в бесконечный пустой цикл.

#### Определение: volatile sig\_atomic\_t

`volatile` — a qualifier that forces the compiler to always read the value from memory, forbidding caching in registers. `sig_atomic_t` — an integer type that guarantees atomicity of read and write operations even if the process is interrupted by a signal. On most architectures, this is an `int` aligned to the word boundary.

## 16 Манипуляция масками сигналов

Для управления моментом доставки сигналов используется маска заблокированных сигналов процесса. Заблокированный сигнал не игнорируется, а переходит в состояние *pending* и доставляется сразу после разблокировки.

```

1 sigset\_\_t set;
2 sigemptyset(&set);
3 sigaddset(&set, SIGINT);

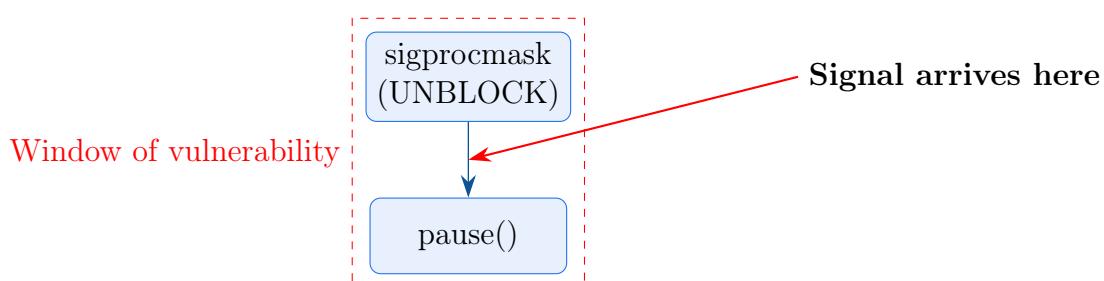
4
5 // Block SIGINT
6 sigprocmask(SIG\_BLOCK, &set, &old\_\_mask);
7 // Critical section: SIGINT delivery is deferred
8 sigprocmask(SIG\_SETMASK, &old\_\_mask, NULL);

```

**Листинг 6 – Signal mask manipulation**

## 17 Критическая гонка (Race Condition): pause()

Исторически для ожидания сигнала использовался вызов `pause()`, который приостанавливает поток до получения любого сигнала. Попытка реализовать безопасное ожидание через разблокировку и `pause()` порождает классическую ошибку Race Condition.



**Рис. 4 – Race condition between signal unblocking and entering sleep state**

**Примечание**

Если сигнал доставляется в зазор между `sigprocmask` и `pause()`, обработчик выполнится немедленно. После этого `pause()` уснет навсегда, так как единственный ожидаемый сигнал уже был обработан.

## 18 Решение: системный вызов `sigsuspend`

Для устранения гонки необходимо обеспечить атомарность двух действий: временной подмены маски сигналов и перевода процесса в состояние ожидания.

**Определение: `sigsuspend`**

An atomic system call that: 1. Sets a new temporary signal mask. 2. Suspends the process until an unblocked signal arrives. 3. Restores the original signal mask upon return.

```

1 volatile sig_atomic_t done = 0;
2 void handler(int s) { done = 1; }

3
4 int main() {
5     sigset(SIG_BLOCK, &mask);
6     sigemptyset(&mask);
7     sigaddset(&mask, SIGINT);
8     // 1. Block signal in advance
9     sigprocmask(SIG_BLOCK, &mask, &wait);
10
11    signal(SIGINT, handler);
12
13    while (!done) {
14        // 2. Atomically unblock and wait
15        if (sigsuspend(&wait) == -1)
16            break;
17
18        // 3. Restore original signal mask
19        if (sigprocmask(SIG_SETMASK, &wait, NULL) != 0)
20            break;
21    }
}

```

Листинг 7 – Safe signal waiting pattern

## 19 Итоги раздела

**Итоги раздела**

- **Busy wait** недопустим из-за неэффективности и непредсказуемости оптимизаций компилятора.
- Использование `volatile sig_atomic_t` обязательно для флагов, изменяемых в обработчиках.
- `sigprocmask` позволяет откладывать доставку сигналов, переводя их в состояние pending.
- Вызов `sigsuspend` — единственный надежный способ ожидания конкретного

сигнала, исключающий потерю уведомления в критическом интервале между разблокировкой и системным вызовом ожидания.

## 20 Контроль контекста и современные рантаймы: `sigaction` и `Go Preemption`

Завершающим этапом изучения механизмов обработки сигналов является переход от упрощенного интерфейса `signal()` к профессиональному стандарту `sigaction`. Этот интерфейс предоставляет полный контроль над состоянием процесса в момент прерывания, позволяя не только обрабатывать ошибки, но и реализовывать сложные механизмы управления рантаймами высокоуровневых языков.

## 21 Интерфейс `sigaction`: Преимущества и флаги

Системный вызов `sigaction()` является предпочтительным в современных POSIX-системах, так как он гарантирует предсказуемое поведение масок сигналов и позволяет настраивать семантику прерываний через структуру `struct sigaction`.

### Определение: Reentrancy (Реентерабельность)

Свойство функции или участка кода, позволяющее его безопасный повторный вызов до завершения предыдущего вызова. В контексте сигналов это означает, что функция не должна использовать статические или глобальные неблокируемые ресурсы, которые могут быть повреждены при внезапном прерывании.

### 21.1 Флаг `SA_RESTART` и обработка `EINTR`

Одной из главных сложностей при работе с сигналами является прерывание «медленных» системных вызовов (например, `read()` из сокета или `wait()`).

#### Примечание

Если сигнал доставляется во время выполнения системного вызова, ядро может либо вернуть ошибку `EINTR`, либо автоматически перезапустить вызов после завершения обработчика. Поведение по умолчанию зависит от версии ОС, поэтому флаг `SA_RESTART` используется для принудительного включения автоматического перезапуска.

## 22 Расширенная информация: `SA_SIGINFO` и `siginfo_t`

При установке флага `SA_SIGINFO` обработчик сигнала принимает три аргумента вместо одного. Это дает доступ к структуре `siginfo_t`, содержащей метаданные о причине возникновения сигнала.

```

1 void segfault\_handler(int sig, siginfo\_t *si, void *unused) {
2     // si\_addr contains the memory address that triggered the MMU exception
3     write(STDERR\_FILENO, "Segmentation Fault at address: ", 31);
4     // In production code, address-to-HEX conversion (signal-safe) would follow
5     \_exit(EXIT\_FAILURE);
6 }
7
8 int main() {
9     struct sigaction sa;

```

```

10     sa.sa\_sigaction = segfault\_handler;
11     sigemptyset(&sa.sa\_mask);
12     sa.sa\_flags = SA\_SIGINFO; // Enable extended signal handler mode
13
14     sigaction(SIGSEGV, &sa, NULL);
15     int *p = NULL;
16     *p = 42; // Triggers SIGSEGV
17     return 0;
18 }
```

Листинг 8 – SIGSEGV processing using SA\_SIGINFO

## 23 Низкоуровневая манипуляция контекстом: ucontext\_t

Третий аргумент обработчика сигнала (`void *ucontext`) в действительности является указателем на структуру `ucontext_t`. Она содержит полное состояние регистров процессора на момент прерывания.

- **REG\_RIP:** Указатель на следующую инструкцию.
- **REG\_RSP:** Текущий указатель стека.
- **REG\_RAX / REG\_RBX / ...:** Значения регистров общего назначения.

Модифицируя эти значения внутри обработчика, программа может принудительно изменить точку возврата из прерывания, что является основой для реализации кооперативной и вытесняющей многозадачности в пользовательском пространстве.

## 24 Case Study: Вытеснение в языке Go (Preemption)

Рантайм языка Go использует сигналы для реализации вытесняющей многозадачности (Preemptive Multitasking). Еслиgorутина выполняется слишком долго без блокирующих вызовов, планировщик Go отправляет потоку сигнал (обычно SIGURG).

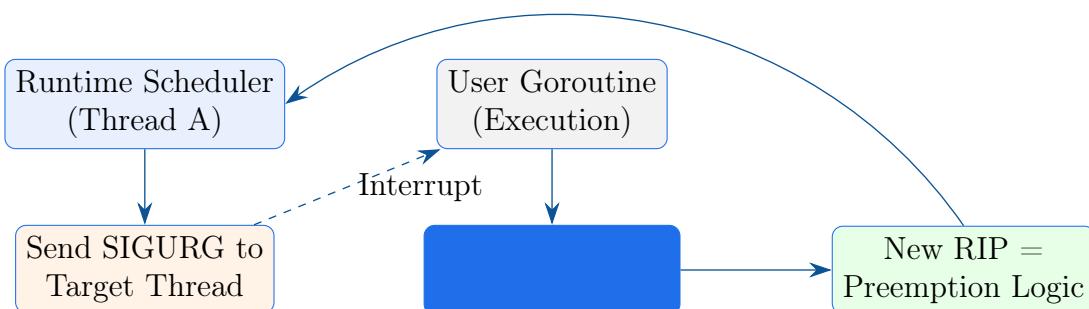


Рис. 5 – Механизм вытеснения горутины через манипуляцию RIP

Обработчик сигнала в рантайме Go анализирует сохраненный `ucontext_t`. Если прерывание произошло в безопасной точке, он подменяет значение `REG_RIP` в структуре на адрес функции планировщика. Когда ядро восстанавливает контекст, поток оказывается не в прерванном коде, а в процедуре переключения горутин.

## 25 Итоги раздела

### Итоги раздела

- **sigaction** — золотой стандарт работы с сигналами, предотвращающий Race Conditions и дающий доступ к метаданным события.
- Флаг **SA\_RESTART** избавляет разработчика от необходимости вручную обрабатывать ошибки прерванных вызовов (**EINTR**).
- **SA\_SIGINFO** позволяет диагностировать причины падения (**si\_addr**) и восстанавливать цепочку вызовов.
- Манипуляция **ucontext\_t** превращает механизм сигналов в мощный инструмент управления потоком исполнения, используемый в рантаймах современных языков для вытеснения задач и реализации высокоуровневой абстракции «горутин».