

ОПЕРАЦИОННЫЕ СИСТЕМЫ • ЛЕКЦИЯ 6

Методы обеспечения многозадачности в ОС

Процессы, потоки, волокна.

Межпроцессное взаимодействие. Синхронизация

1. Многозадачность: основные понятия
2. Процесс: модель, состояния, PCB
3. Жизненный цикл: fork, exec, wait, exit
4. Процессы в Windows
5. Планирование процессора
6. Планировщики Linux и Windows
7. Потоки (threads)
8. Волокна (fibers) и сопрограммы
9. Межпроцессное взаимодействие (IPC)
10. Каналы и сигналы
11. Разделяемая память, очереди, сокет
12. Гонка и критическая секция
13. Прimitives синхронизации
14. Классические задачи
15. Взаимоблокировки (deadlock)
16. Инструменты, практика

1. Многозадачность: основные понятия

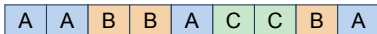
Что такое многозадачность?

Определение

Многозадачность (multitasking) — свойство ОС выполнять несколько задач так, что каждая продвигается вперёд, даже если процессор один.

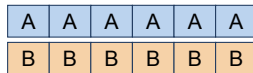
- Ядер обычно 4–16. Задач — **сотни**.
- ОС быстро переключает процессор и создаёт **иллюзию одновременности**.

1 ядро:



Конкурентность — чередование

2 ядра:



Параллелизм — действительно одновременно

История. Кооперативная и вытесняющая многозадачность

Этап	Идея	Решаемая проблема
Пакетная обработка	Задачи строго по очереди	Ручная загрузка программ
Мультипрограммирование	Пока одна ждёт I/O, работает другая	Простой CPU при I/O
Разделение времени	Каждой задаче короткий квант	Интерактивность
Многоядерность	Настоящий параллелизм	Рост производительности без роста частоты

Кооперативная

Задача **сама** отдаёт CPU (yield, I/O). Один бесконечный цикл — зависает вся система. Windows 3.x, Mac OS до 9.

Вытесняющая

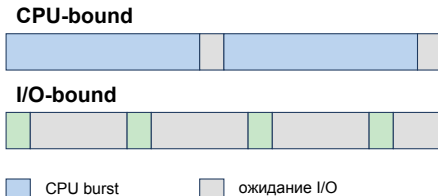
ОС отбирает CPU по **прерыванию таймера**, когда квант истёк. Linux, Windows NT+, macOS.

Задачи CPU-bound и I/O-bound

- **CPU-bound** — много вычисляют (сжатие видео, компиляция): длинные всплески CPU
- **I/O-bound** — много ждут (редактор, веб-сервер): короткие всплески CPU

Правило

Хороший планировщик отдаёт приоритет I/O-bound задачам: они быстро освобождают CPU, а пользователь получает быстрый отклик.

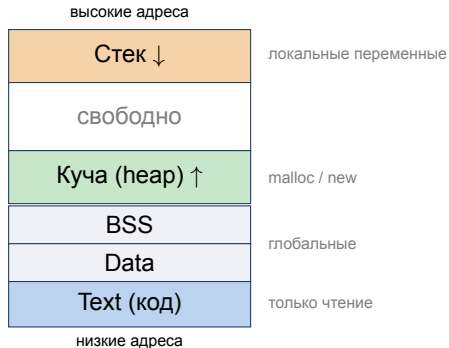


2. Процесс: модель, состояния, РСВ

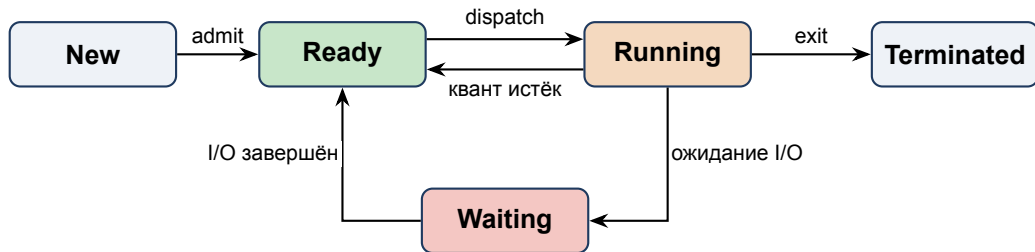
Определение

Процесс — программа в состоянии выполнения вместе с ресурсами от ОС: адресное пространство, открытые файлы, учётные данные, регистры.

- **Программа** — пассивный файл на диске (/usr/bin/python3)
- **Процесс** — активная сущность; 3 терминала = 3 процесса bash
- У каждого процесса **своё виртуальное адресное пространство** — изоляция



Модель состояний процесса



- **Running** → **Ready** — вытеснен, *мог бы* работать дальше
- **Running** → **Waiting** — сам не может продолжать, ждёт данных
- **Waiting** → **Running** напрямую **НЕВОЗМОЖЕН** — сначала в Ready
- На одном ядре в Running не более **одного** процесса

STAT в ps

R	Running или Ready
S	Прерываемый сон
D	Непрерываемый сон (I/O)
T	Остановлен (Ctrl+Z)
Z	Зомби

```
cat /proc/$$/status | head
ls -l /proc/$$/fd
```

PCB (Process Control Block)

Linux: task_struct; Windows: EPROCESS

Идентификатор	PID, PPID
Состояние	Ready, Running...
Контекст CPU	PC, регистры, указатель стека
Планирование	Приоритет, nice, время CPU
Память	Таблицы страниц
Файлы	Таблица дескрипторов
Безопасность	UID, GID, capabilities

3. Жизненный цикл процесса: `fork`, `exec`, `wait`, `exit`

fork() — ОДИН ВЫЗОВ, ДВА ВОЗВРАТА

```
int x = 5;
pid_t pid = fork();
if (pid == -1) { perror("fork"); return 1; }
if (pid == 0) {                      // потомок
    x += 10;
    printf("child: x=%d\n", x);
    exit(0);
}
wait(NULL);                          // родитель ждёт
printf("parent: x=%d\n", x);
```

```
child: x=15
parent: x=5
```

Где	Возвращает
В родителе	PID потомка (> 0)
В потомке	0
При ошибке	-1

Почему у родителя $x = 5$?

После `fork()` у потомка **своя копия** памяти. Изменения в одном не видны в другом.

Copy-on-Write

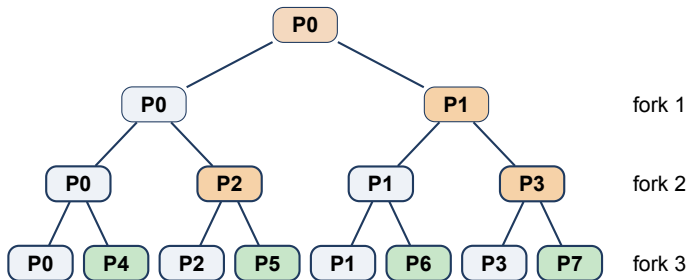
Страницы копируются только **при записи**. Если потомок сразу вызывает `exec()`, почти ничего не копируется.

Подсчёт процессов: n вызовов `fork` $\Rightarrow 2^n$

```
fork();  
fork();  
fork();  
printf("salom\n");
```

Формула

n последовательных `fork()`
 $\Rightarrow 2^n$ процессов
(новых: $2^n - 1$)



8 процессов $\Rightarrow$ 8 строк ``salom``

&& и || вычисляют правую часть не всегда

```
if (fork() && fork())  
    fork();  
printf("x\n");    // 4 процесса
```

Второй `fork()` выполняет только **родитель**: у потомка первый `fork()` вернул 0, и `&&` останавливается.

Ловушка буферизации

`printf` пишет в буфер. Если вывод в канал или файл, буфер **копируется** при `fork()` — строки дублируются.

```
fflush(stdout);    // до fork  
setvbuf(stdout, NULL, _IONBF, 0); // или
```

exec() и как оболочка запускает команду

```
pid_t pid = fork();
if (pid == 0) {
    execlp("ls", "ls", "-l", "/tmp", NULL)
    ;
    perror("execlp"); // только при
    ошибке
    _exit(127);
}
int status;
waitpid(pid, &status, 0);
if (WIFEXITED(status))
    printf("ls exited %d\n",
        WEXITSTATUS(status));
```

- exec **заменяет** код, данные и стек
- **PID сохраняется**, открытые файлы тоже
- При успехе exec **не возвращается**



Зомби и сироты

Ситуация	Название	Что происходит
Потомок завершился, родитель не вызвал <code>wait</code>	Зомби (Z)	Остаётся только запись: PID и код возврата. CPU и память не тратит, но занимает PID
Родитель завершился раньше потомка	Сирота	Потомка усыновляет <code>systemd</code> (PID 1) и сам вызывает <code>wait</code>

Зомби нельзя убить

Он уже мёртв! `kill -9` не поможет. Нужен `wait` в родителе или завершение родителя.

Чем опасно?

Если сервер создаёт тысячи потомков без `wait`, таблица процессов заполняется — новый процесс не создать.

4. Процессы в Windows

CreateProcess и сравнение с POSIX

```
STARTUPINFO si = { sizeof(si) };  
PROCESS_INFORMATION pi;  
CreateProcess(L"C:\\...\\notepad.exe",  
             NULL, NULL, NULL, FALSE, 0,  
             NULL, NULL, &si, &pi);  
WaitForSingleObject(pi.hProcess,  
                    INFINITE);  
CloseHandle(pi.hThread);  
CloseHandle(pi.hProcess);
```

```
Get-Process, tasklist, taskkill /PID  
4312 /F
```

	Linux	Windows
Создание	fork + exec	CreateProcess (1 шаг)
Ожидание	waitpid	WaitForSingleObject
Идентификация	PID	PID + HANDLE
Иерархия	Строгое дерево	Слабая
Зомби	До wait	Пока открыт HANDLE
Группировка	cgroups	Job objects
Единица планирования	Поток (task)	Поток

5. Планирование процессора

Планировщик выбирает процесс из очереди Ready; **диспетчер** переключает.

Когда решение:

1. Running → Waiting
2. Running → Ready (таймер)
3. Waiting → Ready (I/O завершён)
4. Процесс завершился

Только 1, 4 — **невытесняющее**; также 2, 3 — **вытесняющее**.

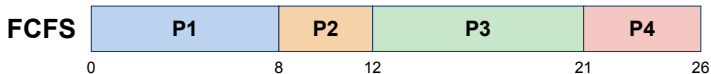
Критерий	Определение	
Загрузка CPU	Доля времени, когда CPU занят	↑
Throughput	Завершённых в единицу времени	↑
Время оборота	Завершение — Поступление	↓
Время ожидания	Оборот — Burst	↓
Время отклика	1-й запуск — Поступление	↓

Алгоритм	Принцип	Вытесн.?	+	—
FCFS	Кто раньше пришёл	Нет	Просто	Эффект конвоя
SJF	Кратчайший burst	Нет	Мин. среднее ожидание (среди невытесняющих)	Длительность неизвестна; голодание
SRTF	Вытесняющий SJF	Да	Мин. среднее ожидание вообще	Много переключений, голодание
Приоритетный	Наивысший приоритет	Оба	Важные быстрее	Голодание → старение (aging)
RR	По кругу, квант q	Да	Хороший отклик, без голодания	Большое среднее время оборота
MLFQ	Несколько очередей, понижение	Да	Сам выделяет I/O-bound	Сложная настройка

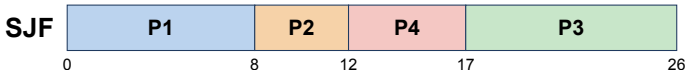
Пример: FCFS и SJF

	Приход	Burst
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Ожидание = Старт –
Приход
(для невытесняющих)



Ожидание: $0 + 7 + 10 + 18 = 35$; **среднее = 8,75**



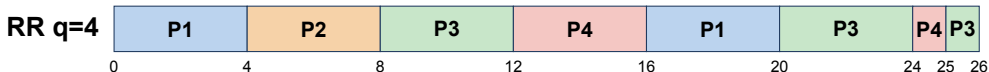
В момент 0 есть только P1. В 8: P2 (4) < P4 (5) < P3 (9)

Ожидание: $0 + 7 + 9 + 15 = 31$; **среднее = 7,75**

Пример: SRTF и RR (q = 4)



В 1 приходит P2 (4): у P1 осталось $7 > 4 \Rightarrow$ P1 вытесняется. Ожидание = Оборот – Burst: P1 = $17 - 8 = 9$, P2 = $4 - 4 = 0$, P4 = $7 - 5 = 2$, P3 = $24 - 9 = 15$. **Среднее = 6,5**



Ожидание: P1 = 12, P2 = 3, P3 = 15, P4 = 17. **Среднее = 11,75**

	FCFS	SJF	SRTF	RR
Среднее ожидание	8,75	7,75	6,5	11,75
Среднее время оборота	15,25	14,25	13,0	18,25

Вывод

SRTF — минимальное среднее ожидание, но нужно знать будущее.

RR — проигрывает по среднему ожиданию, но никто не ждёт следующего кванта дольше $(n - 1) \cdot q$. Именно это нужно интерактивным системам.

Квант q :

- **слишком большой** $\rightarrow$ RR = FCFS, плохой отклик
- **слишком маленький** $\rightarrow$ время уходит на переключения
- на практике: намного больше переключения (микросекунды); обычно 1–100 мс

6. Планировщики Linux и Windows

- **CFS** (2.6.23, 2007): задача с наименьшим `vruntime`; красно-чёрное дерево
- **EEVDF** (6.6, 2023): справедливость + требования к задержке
- **nice**: -20 (высокий) ... $+19$ (низкий); уменьшить может только `root`
- Реальное время: `SCHED_FIFO`, `SCHED_RR`, приоритет 1–99

```
nice -n 10 ./long_job
renice -n 5 -p 4312
chrt -p 4312
taskset -c 0 ./job      # CPU affinity
```

Windows

Потоки по **32 уровням**:

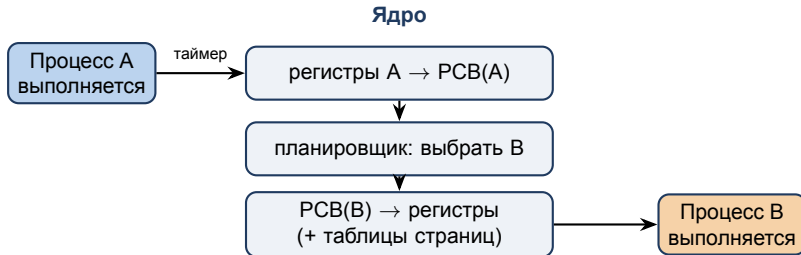
0 — поток обнуления страниц

1–15 — динамические (обычные)

16–31 — реального времени

Внутри уровня — вытесняющий RR.

Активное окно и поток, дождавшийся ввода, — **priority boost**.



Накладные расходы

Полезной работы нет; кэши и TLB «остывают». Между потоками одного процесса **дешевле**: таблицы страниц не меняются.

```
grep ctxt /proc/stat
grep ctxt_switches /proc/$$/status
vmstat 1 5          # столбец cs
```

voluntary — сам заблокировался;
nonvoluntary — вытеснен

7. Потоки (threads)

Поток: что общее, что своё



Общее	Своё
Код	PC
Глобальные, куча	Регистры
Открытые файлы	Стек
PID, UID	TID
Обработчики сигналов	Маска сигналов, errno
	TLS

Зачем нужны потоки? Процесс или поток?

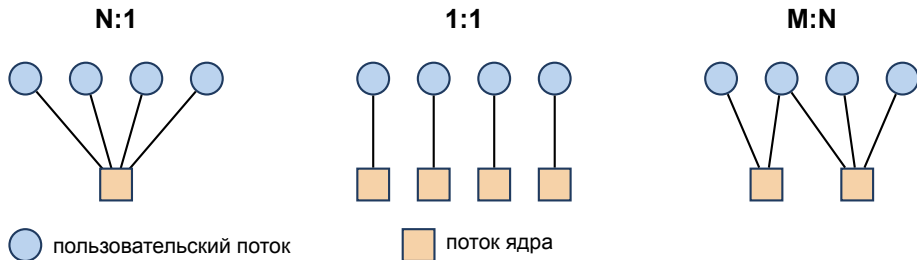
1. **Отзывчивость** — GUI отвечает, рабочий поток загружает
2. **Разделение** — IPC не нужен
3. **Экономия** — создание и переключение дешевле
4. **Масштаб** — параллельно на ядрах

Цена

Нет изоляции: ошибка одного потока убивает весь процесс; нужна **синхронизация**.

	Процесс	Поток
Адресное пространство	Своё	Общее
Создание	Дорого	Дёшево
Переключение	Дороже	Дешевле
Обмен данными	IPC	Напрямую
При сбое	Другие не страдают	Падает весь процесс
Пример	Вкладки Chrome	Потоки веб-сервера

Модели реализации потоков



	N:1	1:1	M:N
+	Очень быстрое переключение	Настоящий параллелизм	Сочетает достоинства
—	Блокирующий вызов останавливает все ; нет параллелизма	Создание дорожки	Сложно
Пример	Старые green threads Java	Linux NPTL, Windows	Go, виртуальные потоки Java 21

Потоки в Linux и pthreads

В Linux и процесс, и поток — **task**. Разница — во флагах `clone()`:

```
fork() ~ clone(SIGCHLD)
pthread_create() ~ clone(CLONE_VM |
    CLONE_FILES | CLONE_THREAD | ...)
```

У потоков одинаковый PID (TGID), разные TID.

```
ps -eLf | head          # LWP = TID
ls /proc/<PID>/task
top -H -p <PID>
```

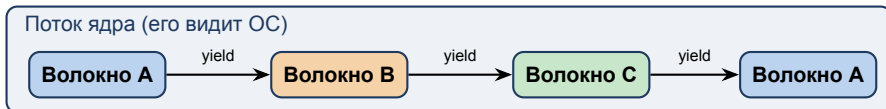
```
void *worker(void *arg) {
    int id = *(int *)arg;
    printf("worker %d started\n", id);
    return NULL;
}

int main(void) {
    pthread_t t[3]; int ids[3] = {1,2,3};
    for (int i = 0; i < 3; i++)
        pthread_create(&t[i], NULL,
                       worker, &ids[i]);
    for (int i = 0; i < 3; i++)
        pthread_join(t[i], NULL);
}
```

Порядок строк **не определён** — его выбирает планировщик.

8. Волокна (fibers) и сопрограммы

Волокно: лёгкий кооперативный «поток»



	Поток (thread)	Волокно (fiber)
Кто планирует	Ядро ОС	Программа (библиотека)
Тип	Вытесняющий	Кооперативный
Цена переключения	Системный вызов, микросекунды	Несколько регистров, наносекунды
Параллелизм	Да	Нет внутри одного потока
Блокирующий вызов	Блокирует один поток	Блокирует все волокна потока
Количество	Тысячи	Миллионы

Windows: ConvertThreadToFiber, CreateFiber, SwitchToFiber. POSIX: swapcontext (устарел).

Сопрограмма — функция, которая приостанавливается и потом продолжает с того же места.

- `async/await`: Python, C#, JS, Rust
- Горутины Go (M:N)
- Виртуальные потоки Java 21

Где полезны

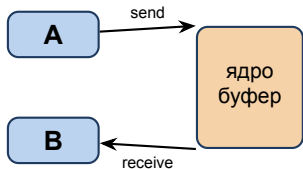
Серверы с огромным числом соединений, которые ждут сеть: поток ядра на каждое — слишком дорого.

	Процесс	Поток	Волокно
Изоляция памяти	Да	Нет	Нет
Кто планирует	ОС	ОС	Программа
Вытеснение	Да	Да	Нет
Параллелизм	Да	Да	Через потоки
Стоимость	Высокая	Средняя	Низкая

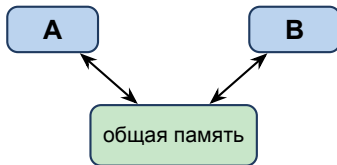
9. Межпроцессное взаимодействие (IPC)

Две модели IPC

Передача сообщений



Разделяемая память



Плюсы

Минусы

Передача сообщений

Просто, ядро синхронизирует, работает и по сети

Каждое сообщение копируется через ядро

Разделяемая память

Максимальная скорость, без копирования

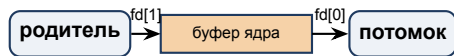
Синхронизация — забота программиста

Механизм	Направление	Родство	Сеть	Особенность
Анонимный канал	Одностороннее	Только родственные	Нет	Поток байтов,
FIFO	Одностороннее	Любые	Нет	Имя в файловой системе
Сигналы	Уведомление	Любые	Нет	Только номер
Очередь сообщений	Двустороннее	Любые	Нет	Границы, приоритеты
Разделяемая память	Общая	Любые	Нет	Быстрее всех, нужна синхронизация
Unix-сокет	Двустороннее	Одна машина	Нет	Передача дескрипторов
Сокет TCP/UDP	Двустороннее	Любые	Да	Универсальный
Файл mmap	Общая	Любые	Нет	Память, связанная с файлом

10. Каналы (pipe, FIFO) и сигналы

Анонимный канал (pipe)

```
int fd[2];          // 0: чтение, 1: запись
pipe(fd);
pid_t pid = fork();
if (pid == 0) {      // потомок
    close(fd[1]);
    char buf[64];
    ssize_t n = read(fd[0], buf, 63);
    buf[n] = '\0';
    printf("child got: %s\n", buf);
    return 0;
}
close(fd[0]);        // родитель
write(fd[1], "salom, bola!", 12);
close(fd[1]);        // -> EOF
wait(NULL);
```



- read из пустого канала — **блокируется**
- write в полный — блокируется (Linux: 64 КиБ)
- Все писатели закрыты — read = 0 (**EOF**)
- Запись без читателя — SIGPIPE

Типичная ошибка

Не закрыть ненужный конец: если потомок не закроет `fd[1]`, он никогда не получит EOF и зависнет.

Именованный канал (FIFO)

```
mkfifo /tmp/myfifo
ls -l /tmp/myfifo      # prw-r--r-- : тип 'p'
# Терминал 1:
cat /tmp/myfifo        # ждёт писателя
# Терминал 2:
echo "salom" > /tmp/myfifo
rm /tmp/myfifo
```

Отличие

Анонимным каналом пользуются только **родственные** процессы. У FIFO есть **имя** в файловой системе — его найдёт любой процесс.

Важно

Данные **не пишутся на диск** — имя нужно только, чтобы найти канал.

Каждый день: `ps aux | grep python | wc -l`

Сигнал	№	По умолчанию	Когда
SIGINT	2	Завершение	Ctrl+C
SIGKILL	9	Завершение	не перехватить
SIGSEGV	11	Завершение + дамп	Неверная память
SIGPIPE	13	Завершение	Канал без читателя
SIGTERM	15	Завершение	Вежливая просьба
SIGCHLD	17*	Игнорируется	Потомок завершился
SIGCONT	18*	Продолжение	fg, bg
SIGSTOP	19*	Остановка	не перехватить
SIGTSTP	20*	Остановка	Ctrl+Z

* номера Linux x86/ARM; в скриптах используйте имена: `kill -TERM`

Сигнал — **асинхронное уведомление**; только **номер**, без данных.

Процесс выбирает: действие по умолчанию, игнорирование или **обработчик**.

Сначала TERM, потом KILL

SIGTERM даёт сохранить данные и закрыть файлы. SIGKILL — мгновенно, данные могут потеряться.

```
volatile sig_atomic_t stop = 0;

void on_sigint(int sig) { stop = 1; }

int main(void) {
    struct sigaction sa = {0};
    sa.sa_handler = on_sigint;
    sigaction(SIGINT, &sa, NULL);
    while (!stop) pause();
    printf("SIGINT received\n");
}
```

Правило

В обработчике только **async-signal-safe** функции. `printf`, `malloc` — **нельзя**.
Только установить флаг.

```
kill -l
kill -TERM 4312
kill -9 4312          # последнее
                     средство
kill -STOP 4312; kill -CONT 4312
trap 'echo поймано' INT
```

11. Разделяемая память, очереди, сокеты

```
// Писатель
int fd = shm_open("/demo_shm",
                  O_CREAT | O_RDWR, 0600);
ftruncate(fd, 4096);
char *p = mmap(NULL, 4096,
               PROT_READ | PROT_WRITE,
               MAP_SHARED, fd, 0);
strcpy(p, "salom from shared memory");

// Читатель
int fd = shm_open("/demo_shm", O_RDONLY, 0);
char *p = mmap(NULL, 4096, PROT_READ,
               MAP_SHARED, fd, 0);
printf("read: %s\n", p);
shm_unlink("/demo_shm");
```

Одна и та же физическая память отображается в оба адресных пространства — **самый быстрый** IPC.

Синхронизации нет!

Читатель может увидеть недописанные данные. Используйте семафор (`sem_open`) или process-shared мьютекс.

<code>ls -l /dev/shm</code>	# объекты <i>POSIX</i>
<code>ipcs -a</code>	# объекты <i>System V</i>

Очередь сообщений

Сохраняются **границы** сообщений и **приоритеты** (канал — поток байтов).

```
mqd_t q = mq_open("/demo_mq",  
                  O_CREAT | O_RDWR, 0600, NULL);  
mq_send(q, "task-42", 8, 5); // prio 5  
mq_receive(q, buf, sizeof buf, &prio);  
mq_unlink("/demo_mq");
```

Сокеты

Один интерфейс: `socket`, `bind`, `listen`, `accept`, `connect`, `send`, `recv`

- **локально** — `AF_UNIX` (Docker, `systemd`)
- **по сети** — `TCP/UDP`

```
ss -xl | head      # Unix сокеты-  
ss -tlnp           # TCP + процессы
```

Единственный

Только сокеты работают между **разными компьютерами**.

Windows	Аналог в Linux
Анонимные каналы (CreatePipe)	pipe()
Именованные каналы (\\.\pipe\name)	FIFO, но двусторонние и по сети
CreateFileMapping + MapViewOfFile	shm_open + mmap
Mailslots	Широковещательные дейтаграммы
Winsock	Сокеты BSD
COM / RPC / ALPC	D-Bus, RPC
Сигналов нет; события (events)	Сигналы

12. Гонка и критическая секция

Состояние гонки (race condition)

```
#define N 1000000
long counter = 0;           // общая

void *worker(void *arg) {
    for (int i = 0; i < N; i++)
        counter++;          // НЕ атомарно!
    return NULL;
}
// main: 2 потока, затем join
```

Реальный результат:

```
counter = 1068612 (expected 2000000)
counter = 1049141 (expected 2000000)
```

Определение

Гонка — результат зависит от порядка выполнения операций.

counter++ = **3 шага**:

1. LOAD R ← counter
2. ADD R ← R + 1
3. STORE counter ← R

Каждый раз **другой** результат, меньше ожидаемого.

Потерянное обновление (lost update)

Шаг	Поток А	Поток В	counter
1	LOAD R=5		5
2		LOAD R=5	5
3	ADD R=6		5
4		ADD R=6	5
5	STORE 6		6
6		STORE 6	6 (должно быть 7)

Data race

Два потока обращаются к одной ячейке без синхронизации, и хотя бы один пишет. В C/C++ это — **неопределённое поведение**.



В секции одновременно **не более одного** потока.

1. **Взаимное исключение** — в секции не более одного потока
2. **Прогресс** — если секция свободна, решение о входе принимается за конечное время
3. **Ограниченное ожидание** — пока поток ждёт, другие не могут входить бесконечно (нет голодания)

Простой флаг не работает

`while (busy) {} busy = 1;` — проверка и установка — **две операции**; войти могут оба потока.

Петерсон (2 потока)

```
int flag[2] = {0, 0};
int turn;
void lock(int i) {
    int j = 1 - i;
    flag[i] = 1;    // хочу войти
    turn = j;       // уступаю очередь
    while (flag[j] && turn == j) { }
}
void unlock(int i) { flag[i] = 0; }
```

Выполняет 3 требования, но современный CPU **переупорядочивает** операции — нужны барьеры памяти.

Аппаратная поддержка

- **Test-and-Set** — атомарно прочитать и записать 1
- **Compare-and-Swap** — «если expected, записать new»
- **Fetch-and-Add** — x86: lock xadd

```
#include <stdatomic.h>
atomic_long counter = 0;
atomic_fetch_add(&counter, 1);

atomic_flag lk = ATOMIC_FLAG_INIT;
while (atomic_flag_test_and_set(&lk)) {}
atomic_flag_clear(&lk);
```

13. Примитивы синхронизации

Мьютекс и спин-блокировка

```
pthread_mutex_t m =  
    PTHREAD_MUTEX_INITIALIZER;  
for (int i = 0; i < N; i++) {  
    pthread_mutex_lock(&m);  
    counter++;  
    pthread_mutex_unlock(&m);  
}  
// результат всегда 20000000
```

У **мьютекса** есть **владелец**: открыть может только захвативший поток. Если занят — поток **засыпает**.

	Мьютекс	Spinlock
Ожидание	Сон	Активное (тратит CPU)
Цена	Переключение контекста	Циклы CPU
Когда	Длинная секция, I/O	Очень короткая, многоядерность
Где	Программы	Ядро, прерывания

На одном ядре

Spinlock бессмыслен: пока ждущий крутится, владелец не может выполниться и освободить.

Семафор (Дейкстра, 1965)

S — целый счётчик, 2 атомарные операции:

- **wait(S)** (P): если $S > 0$, то $S--$, иначе блокировка
- **signal(S)** (V): $S++$, разбудить одного ждущего

```
sem_t slots;  
sem_init(&slots, 0, 3);    // 3 места  
sem_wait(&slots);  
use_database();  
sem_post(&slots);
```

Для упорядочивания:

```
sem_t done = 0;  
A: step1(); sem_post(&done);  
B: sem_wait(&done); step2();
```

	Мьютекс	Семафор
Значение	занят / свободен	целое ≥ 0
Владелец	есть	нет
Назначение	исключение	счёт, сигнал

В macOS нет `sem_init` — используйте `sem_open`.

```
pthread_mutex_lock(&m);
while (!condition)           // while!
    pthread_cond_wait(&cv, &m);
/* условие выполнено, m захвачен */
pthread_mutex_unlock(&m);

// другой поток:
pthread_mutex_lock(&m);
condition = 1;
pthread_cond_signal(&cv);
pthread_mutex_unlock(&m);
```

`cond_wait` атомарно **отпускает m** и **засыпает**, при пробуждении снова захватывает m.

Почему while?

Ложные пробуждения; другой поток мог «забрать» условие.

- **Монитор** — данные + мьютекс + условия (Java synchronized)
- **RW lock** — много читателей *или* один писатель
- **Барьер** — N потоков ждут друг друга

Выбор примитива. Аналоги в Windows

Задача	Примитив	Windows	POSIX
Простой счётчик/флаг	Атомарная переменная	CRITICAL_SECTION, SRWLOCK	mutex, rwlock
Структура данных	Мьютекс		
Много чтений, мало записей	RW lock	CreateMutex (между процессами)	process-shared mutex
Ограничить число пользователей	Считающий семафор	CreateSemaphore	sem_t
Ждать условия	Усл. перем. + мьютекс	CreateEvent	cond var + флаг
Синхронизировать процессы	Именованный семафор	Interlocked*	stdatomic.h
Короткая секция в ядре	Spinlock	WaitForSingleObject	sem_wait, join

14. Классические задачи синхронизации

Производитель–потребитель

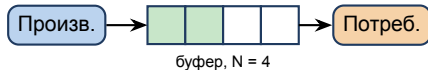
```
semaphore mutex = 1; // исключение  
semaphore empty = N; // свободно  
semaphore full = 0; // занято
```

Производитель:

```
wait(empty);  
wait(mutex);  
put(item);  
signal(mutex);  
signal(full);
```

Потребитель:

```
wait(full);  
wait(mutex);  
item = take();  
signal(mutex);  
signal(empty);
```



Порядок важен!

Если сделать `wait(mutex)` **до** `wait(empty)` при полном буфере — уснёт с мьютексом: **взаимоблокировка**.

В жизни: cmd1 | cmd2, очередь пула потоков, буфер печати.

```
semaphore rw_mutex = 1, mutex = 1;  
int read_count = 0;
```

Писатель:	Читатель:
<code>wait(rw_mutex);</code>	<code>wait(mutex);</code>
<code>write();</code>	<code>read_count++;</code>
<code>signal(rw_mutex);</code>	<code>if (read_count == 1)</code>
	<code>wait(rw_mutex);</code>
	<code>signal(mutex);</code>
	<code>read();</code>
	<code>wait(mutex);</code>
	<code>read_count--;</code>
	<code>if (read_count == 0)</code>
	<code>signal(rw_mutex);</code>
	<code>signal(mutex);</code>

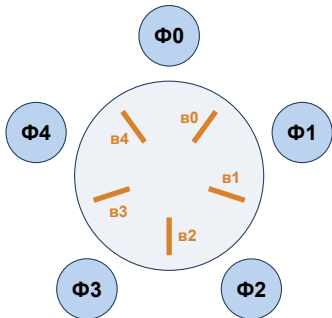
Правило

Читатели — **одновременно**;
писатель — **в одиночку**.

Первый читатель блокирует писателей, последний — впускает.

Голодание

При непрерывном потоке читателей писатель ждёт вечно. Готовое решение: `pthread_rwlock_t`.



5 философов, 5 вилок. Чтобы есть, нужны **обе** соседние вилки.

Наивное решение

«Сначала левую, потом правую» — если все возьмут левую, **взаимоблокировка**.

Решения:

1. **Упорядочивание**: сначала вилка с меньшим номером (Ф4: в0, потом в4)
2. За столом не более **4** философов (семафор = 4)
3. **Асимметрия**: чётные — с левой, нечётные — с правой
4. Брать обе вилки атомарно

15. Взаимоблокировки (deadlock)

Взаимоблокировка и условия Коффмана

```
// Поток 1      // Поток 2
lock(A);        lock(B);
lock(B);        lock(A);
// ждёт B       // ждёт A
```

В жизни

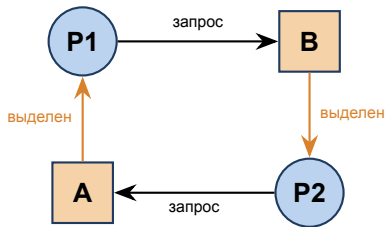
Две машины на узком мосту: никто не сдаёт назад.

Коффман (1971) — все четыре одновременно:

- | | | |
|---|----------------------|--|
| 1 | Взаимное исключение | Ресурсом пользуется один процесс |
| 2 | Удержание и ожидание | Держит и ждёт другой |
| 3 | Нет вытеснения | Отдают только добровольно |
| 4 | Циклическое ожидание | $P1 \rightarrow P2 \rightarrow \dots \rightarrow P1$ |

Достаточно нарушить одно!

Граф распределения ресурсов



Цикл: $P1 \rightarrow B \rightarrow P2 \rightarrow A \rightarrow P1$

- Круг — процесс, квадрат — ресурс
- $P \rightarrow R$ — запрос
- $R \rightarrow P$ — выделен

Правило

У каждого ресурса **один экземпляр**: цикл $\Leftrightarrow$ взаимоблокировка.

Несколько экземпляров: цикл необходим, но **не достаточен**.

Подход	Идея	Пример
Предотвращение	Нарушить одно из 4 условий	Единый порядок захвата ; запрашивать всё сразу
Обход	Выдавать, только если состояние безопасно	Алгоритм банкира
Обнаружение и восстановление	Искать циклы и устранять	Завершить жертву, откатить транзакцию (СУБД)
Игнорирование	«Страусиный алгоритм»	Большинство ОС общего назначения

Самое практичное для программиста

Если все потоки всегда захватывают A раньше B — цикл образоваться не может.

	Allocation	Max	Need
	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1

Available = (3, 3, 2); всего (10, 5, 7)

Идея

ОС знает **максимальную потребность** каждого процесса. Ресурс выдаётся, только если состояние остаётся **безопасным**.

Need = Max – Allocation

Безопасное состояние: есть порядок, в котором все могут завершиться.

Алгоритм банкира: проверка безопасности

Шаг	Процесс	Need $\leq$ Work?	Work += Allocation
1	P1	$(1,2,2) \leq (3,3,2)$ ✓	(5,3,2)
2	P3	$(0,1,1) \leq (5,3,2)$ ✓	(7,4,3)
3	P0	$(7,4,3) \leq (7,4,3)$ ✓	(7,5,3)
4	P2	$(6,0,0) \leq (7,5,3)$ ✓	(10,5,5)
5	P4	$(4,3,1) \leq (10,5,5)$ ✓	(10,5,7)

Результат

Состояние **безопасно**, последовательность: **⟨P1, P3, P0, P2, P4⟩**

При запросе: Request $\leq$ Need; Request $\leq$ Available; условно выделить, проверить; если небезопасно — откатить и ждать. **Небезопасно** $\neq$ **тупик**: просто нет гарантии.

Livelock

Потоки активны, но прогресса нет. Двое в коридоре одновременно уступают друг другу дорогу.

Голодание

Поток бесконечно не получает ресурс — его всё время обходят. Решение: **старение**, справедливые очереди.

Инверсия приоритетов

N ждёт мьютекс L, а L вытесняет M. Решение: **наследование приоритета**. Mars Pathfinder, 1997.

16. Инструменты наблюдения в Linux

Команды наблюдения за процессами

Задача	Команда
Список	<code>ps aux, ps -ef</code>
Дерево	<code>pstree -p</code>
Реальное время	<code>top, htop</code>
Потоки	<code>ps -eLf, top -H</code>
Поиск	<code>pgrep -a, pidof</code>
Сигнал	<code>kill, pkill, killall</code>
Приоритет	<code>nice, renice, chrt</code>
Системные вызовы	<code>strace -f -p <PID></code>
Файлы, каналы	<code>lsof -p, /proc/<PID>/fd</code>
IPC	<code>ipcs -a, /dev/shm, ss -xl</code>
Переключения	<code>vmstat 1, pidstat -w</code>

```
sleep 300 &  
jobs  
ps -o pid,ppid,stat,ni,cmd -p $!  
kill %1
```

```
./pc &  
ps -o pid,nlwp,cmd -p $!  
# мьютексы в Linux = futex  
strace -f -e trace=futex ./race
```

17. Практика и выводы

1. **Дерево процессов:** `ps tree`, PPID для `sleep 100 &`, `/proc/<PID>/status`.
2. **fork и зомби:** `fork1.c`; создать и найти зомби без `wait`.
3. **Подсчёт процессов:** `fork` в цикле и `fork()` || `fork()`.
4. **Гонка:** исправить `race.c` с тремя способами, сравнить время.
5. **Канал и FIFO:** эффект незакрытого `fd[1]`; `wc -l` через FIFO.
6. **Сигналы:** разница INT, TERM, KILL; скрипт с `trap`.
7. **Производитель-потребитель и nice:** `while` → `if`; `nice 0` и `19` на одном ядре.

Среда

Только виртуальная машина Linux: `sudo apt install build-essential strace psmisc`

Ошибка	Последствие	Как правильно
Не вызывать <code>wait()</code>	Зомби, кончаются PID	<code>waitpid</code> или <code>SIGCHLD</code>
Полный буфер перед <code>fork()</code>	Дублирование вывода	<code>fflush(stdout)</code>
Не закрыть конец канала	Нет EOF, зависание	Сразу закрыть ненужный конец
Общие данные без синхронизации	Гонка	Мьютекс, атомарные
<code>if</code> перед <code>cond_wait</code>	Условие не выполнено	<code>while</code>
Блокировки в разном порядке	Взаимоблокировка	Единый порядок
I/O в критической секции	Остальные простаивают	Сократить секцию
Нет <code>unlock</code> на пути ошибки	Вечная блокировка	Одна точка выхода, <code>RAII</code>
<code>volatile</code> вместо синхронизации	Гонка остаётся	<code>atomic_*</code>
<code>printf</code> в обработчике	Блокировка внутри <code>libc</code>	Только флаг
Сразу <code>kill -9</code>	Потеря данных	Сначала <code>SIGTERM</code>

Что нужно запомнить

1. Многозадачность — иллюзия за счёт переключения CPU; вытеснение — **таймер**.
2. **Процесс** — своё адресное пространство; всё о нём — в **PCB**.
3. Состояния: **New, Ready, Running, Waiting, Terminated**.
4. **fork + exec**; без **wait** — зомби.
5. n вызовов `fork()` $\Rightarrow 2^n$ процессов.
6. **SRTF** — мин. ожидание; **RR** — хороший отклик.
7. **Поток** — общая память, свои стек и регистры.
8. **Волокно** — кооперативно, в пространстве пользователя.
9. **IPC**: передача сообщений или разделяемая память.
10. `counter++` не атомарен $\Rightarrow$ **гонка**.
11. Критическая секция: **исключение, прогресс, ограниченное ожидание**.
12. **Deadlock: единый порядок блокировок!**

Контрольные вопросы (1)

1. Чем конкурентность отличается от параллелизма? Возможна ли конкурентность на одном ядре?
2. Какое устройство делает вытесняющую многозадачность возможной?
3. Чем программа отличается от процесса?
4. Почему невозможен переход `Waiting` → `Running`?
5. Что хранится в PCB? Как он называется в Linux?
6. Что возвращает `fork()` в родителе и в потомке?
7. Что такое `copy-on-write`?
8. Сколько процессов будет после четырёх последовательных `fork()`?
9. Что делает `exec()`? Сохраняется ли PID?
10. Что такое зомби и сирота? Можно ли убить зомби командой `kill -9`?
11. Чем `CreateProcess` отличается от `fork + exec`?
12. Дайте определения времени ожидания и времени оборота.
13. Что такое эффект конвоя? Для какого алгоритма он характерен?
14. Почему SJF трудно реализовать на практике?
15. Как размер кванта влияет на RR?

Контрольные вопросы (2)

16. Что такое `vruntime` в CFS? Какой планировщик в Linux с версии 6.6?
17. Почему переключение между потоками дешевле?
18. Что потоки разделяют, а что у каждого своё?
19. Сравните модели N:1, 1:1 и M:N.
20. Чем волокно отличается от потока?
21. Сравните модели «передача сообщений» и «разделяемая память».
22. Почему читатель канала может зависнуть?
23. Разница `SIGTERM` и `SIGKILL`? Какие сигналы нельзя перехватить?
24. Покажите чередование `counter++`, при котором теряется обновление.
25. Три требования к решению задачи критической секции.
26. Чем мьютекс отличается от семафора и от `spinlock`?
27. Почему перед `pthread_cond_wait` используют `while`?
28. Что будет, если поменять местами `wait(empty)` и `wait(mutex)`?
29. Условия Коффмана; какое из них нарушает единый порядок?
30. Безопасно ли состояние из раздела 15? Найдите безопасную последовательность.

1. Таненбаум Э., Бос Х. *Современные операционные системы*. 4-е изд. — СПб.: Питер, 2015. — Гл. 2, 6.
2. Silberschatz A., Galvin P., Gagne G. *Operating System Concepts*. 10th ed. — Wiley, 2018. — Ch. 3–8.
3. Arpaci-Dusseau R., A. *Operating Systems: Three Easy Pieces*. — ostep.org (бесплатно).
4. Kerrisk M. *The Linux Programming Interface*. — No Starch Press, 2010.
5. Downey A. *The Little Book of Semaphores*. — Green Tea Press (бесплатно).
6. Russinovich M. и др. *Windows Internals*. 7th ed., Part 1. — Гл. 3–4.
7. man 2 fork, man 2 wait, man 7 pipe, man 7 signal, man 7 pthreads, man 7 sem_overview, man 7 sched.
8. Microsoft Learn: *Processes and Threads, Synchronization, Interprocess Communications*.

Спасибо за внимание!

Вопросы?