Исключения и процессы

Основы информатики.

Компьютерные основы программирования

goo.gl/X7evF

На основе **CMU** 15-213/18-243:
Introduction to Computer Systems

goo.gl/TDDVV

Лекция **13**, 30 апреля, 2018

Лектор:

Дмитрий Северов, кафедра информатики 608 КПМ

dseverov@mail.mipt.ru

cs.mipt.ru/wp/?page_id=346

Исключения и процессы

- **Поток управления с исключениями**
- **Исключения**
- **Процессы**
- **Управление процессами**

Поток управления

- **Процессор занят только одним:**
  - От запуска до останова, ЦП просто читает и исполняет (интерпретирует) последовательность команд, одну за раз.
  - Эта последовательность – *поток управления ЦП*

Физический поток управления

Время

<запуск>

команда$_1$

команда$_2$

команда$_3$

…

команда$_n$

<останов>

Изменение потока управления

- **До сих пор: два механизма изменения потока управления:**
  - Безусловный и условные переходы
  - Обращение к процедуре и возврат из процедуры

  Оба реагируют на изменения в ***состоянии программы***

- **Недостаточно для практической системы:**
  Трудно реагировать на изменения в *состоянии системы*
  - данные поступают с диска и/или из сети
  - команды делят на ноль
  - пользователь нажимает Ctrl-C на клавиатуре
  - срабатывают системные таймеры

- **Нужны механизмы “потока управления с исключениями”**

Поток управления с исключениями (исключительными ситуациями)

- **Присутствует на всех уровнях компьютерной системы**
- **Низкоуровневые механизмы**
  1. **Исключения**
     - изменения в потоке управления в ответ на события в системе (т.е., на изменение состояния системы)
     - Комбинация аппаратуры и ПО операционной системы

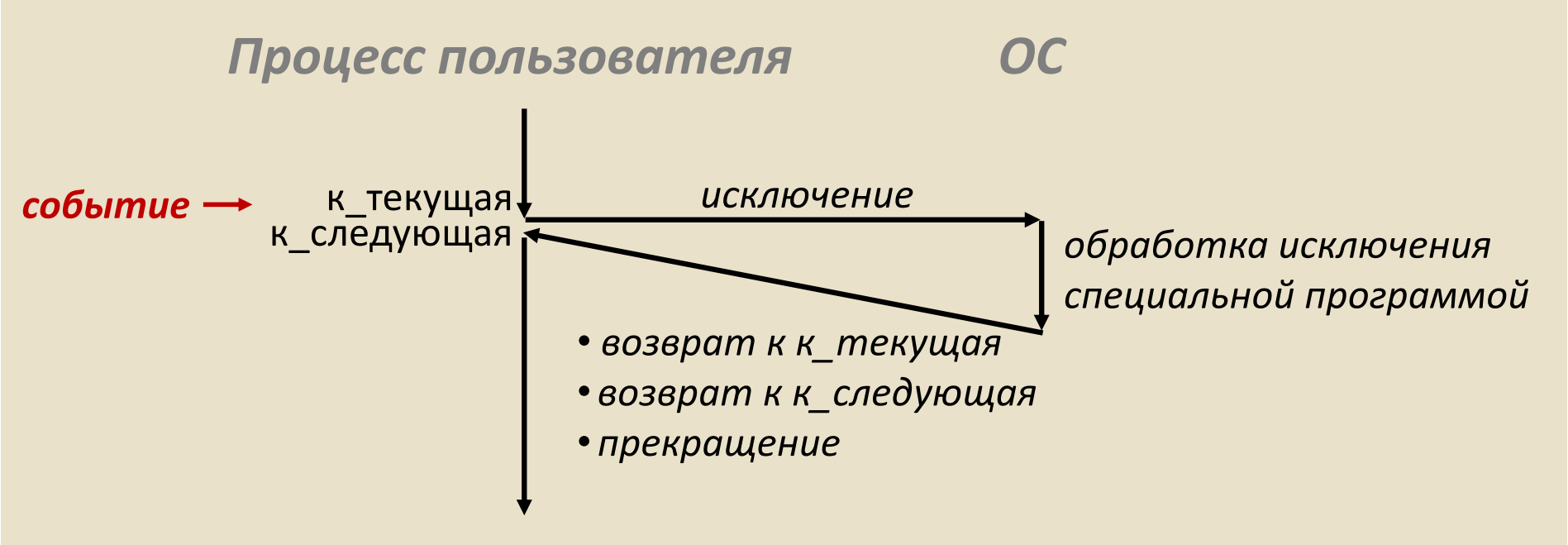
- **Высокоуровневые механизмы**
  2. **Смена контекстов** (вычислительных) процессов
     - ПО операционной системы и аппаратный таймер
  3. **Сигналы**
     - ПО операционной системы
  4. **Нелокальные переходы** : `setjmp()` и `longjmp()`
     - Реализуются средой исполнения языка С

Исключения и процессы

- Поток управления с исключениями
- Исключения
- Процессы
- Управление процессами

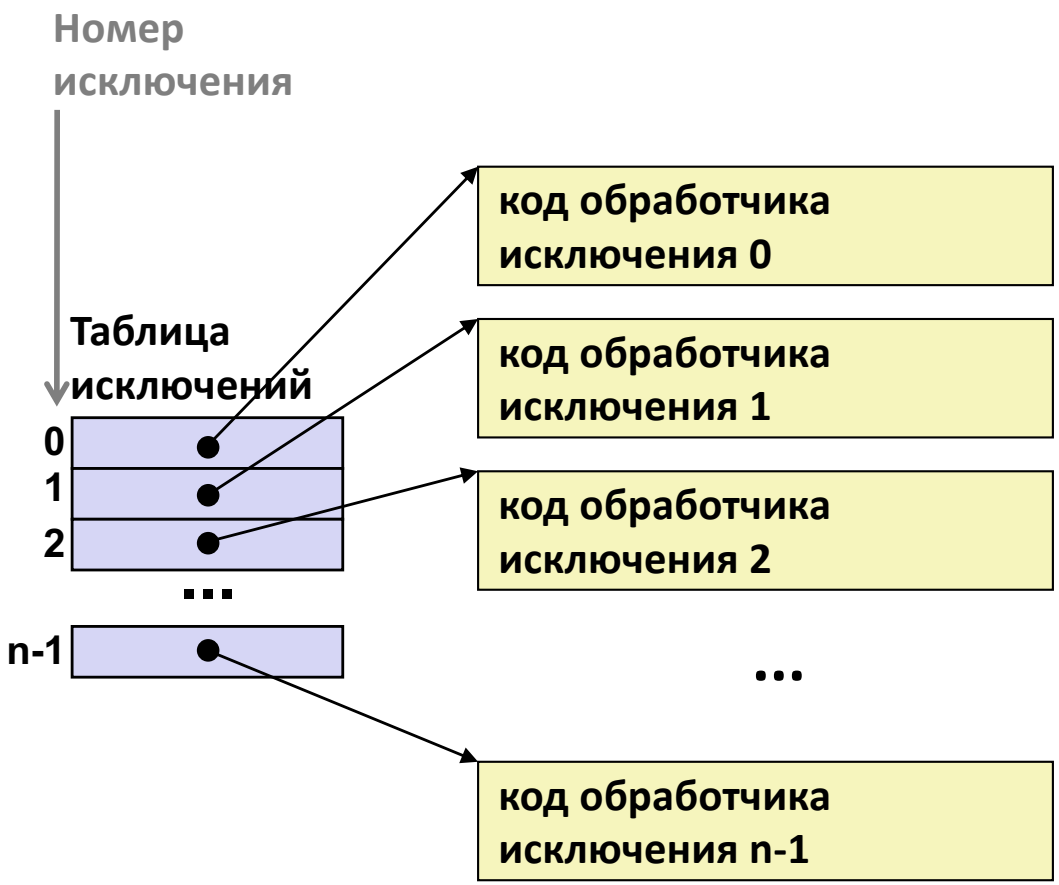
Исключения (исключительные ситуации)

- ***Исключение*** **– передача управления операционной системе в ответ на некоторое *событие* (т.е., изменение состояния ЦП)**



- **Примеры:**
  деление на 0, арифметическое переполнение, ошибочная страница памяти, завершение в/в, Ctrl-C

Вектора прерываний



- Каждый тип событий имеет уникальный номер исключения k
- k = индекс в таблице исключений (т.н. вектор прерывания)
- Обработчик k вызывается всякий раз, как возникает исключение k

Асинхронные исключения (прерывания)

- **Вызываются внешними для процессора событиями**
  - Обозначается установкой напряжения на специальном контакте ЦП
  - Обработчик возвращается к “следующей” команде

- **Примеры:**
  - Прерывание таймера
    - Каждые несколько мс аппаратура таймера делает прерывание
    - Используется ядром ОС для отъёма управления у других программ
  - Ввод/вывод
    - нажатие Ctrl-C на клавиатуре
    - поступление пакета данных из сети
    - поступление блока данных с диска

Синхронные исключения

- **Возникают в результате исполнения команд:**
  - ***Ловушки (Traps)***
    - преднамеренные
    - примеры: ***обращения к ядру ОС***, точки останова, спецкоманды
    - возвращают управление на “следующую” команду
  - ***Сбои (Faults)***
    - непреднамеренные и возможно исправимые
    - примеры: сбой обращения к странице виртуальной памяти (исправима), нарушение защиты виртуальной памяти (неисправима), исключение вычислений с плавающей точкой
    - либо повторно исполняет сбойную (“текущую”) команду или вызывает прекращение работы программы
  - ***Прекращения (Aborts)***
    - непреднамеренные и неисправимые
    - примеры: ошибка чётности памяти, самодиагностика аппаратуры, недопустимая команда
    - прекращает текущую программу

Примеры исключений x86-64

Номер исключения	*Описание*	*Класс исключения*
0	Деление на 0	Сбой
13	Сбой общей защиты	Сбой
14	Страничный сбой	Сбой
18	Самопроверка машины	Прекращение
32-255	Исключения, определяемые ОС	Прерывания, ловушки

Вызовы системы

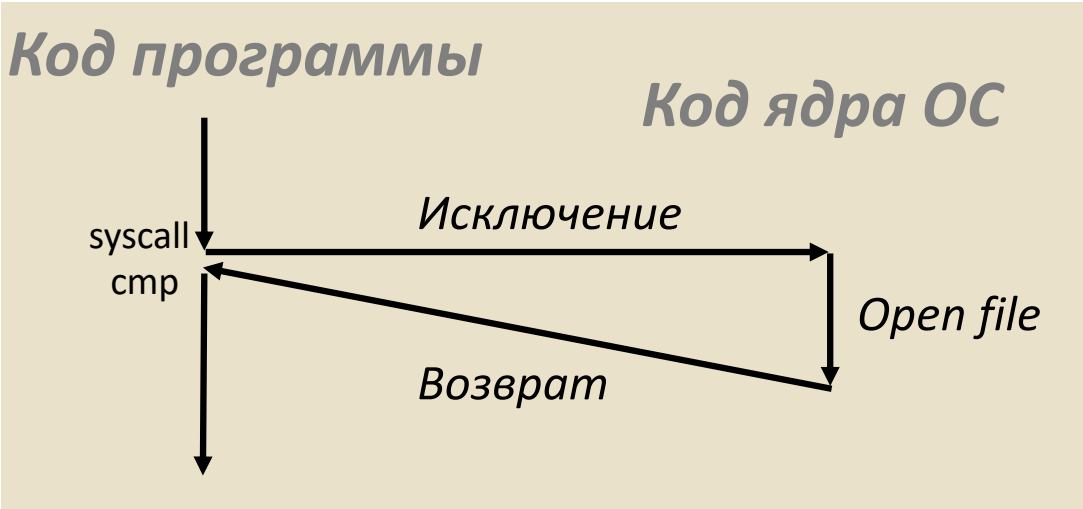
- **Каждое вызов x86-64 системы имеет уникальный номер**
- **Примеры:**

Номер	*Имя*	*Описание*
0	`read`	Прочитать файл
1	`write`	Записать файл
2	`open`	Открыть файл
3	`close`	Закрыть файл
4	`stat`	Получить информацию о файле
57	`fork`	Создать процесс
59	`execve`	Выполнить программу
60	`_exit`	Завершить процесс
62	`kill`	Отправить сигнал процессу

Вызов системы (ловушка): открыть файл

- В С-программе вызвается: **`open(filename, options)`**
- Вызывает функцию **`__open`** с командой системного вызова **`syscall`**

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax   # open – сисвызов №2
e5d7e:   0f 05               syscall          # Результат – в %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



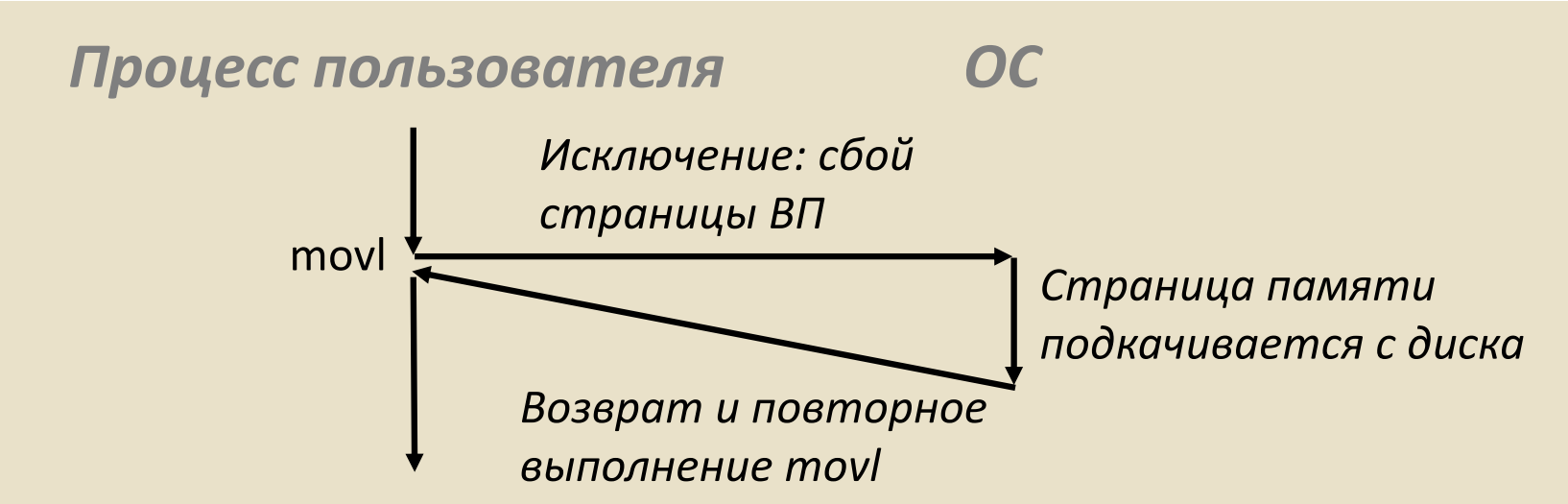
- в `%rax` – номер вызова
- в `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9` – другие аргументы
- Результат в `%rax`
- Отрицательное значение – ошибка, соответсвующаяя отрицательному `errno`

Сбой страницы [виртуальной памяти]

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

- Пользователь пишет в память
- Нужная часть (страница) пользовательской памяти в этот момент откачана на диск

```
80483b7:        c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```

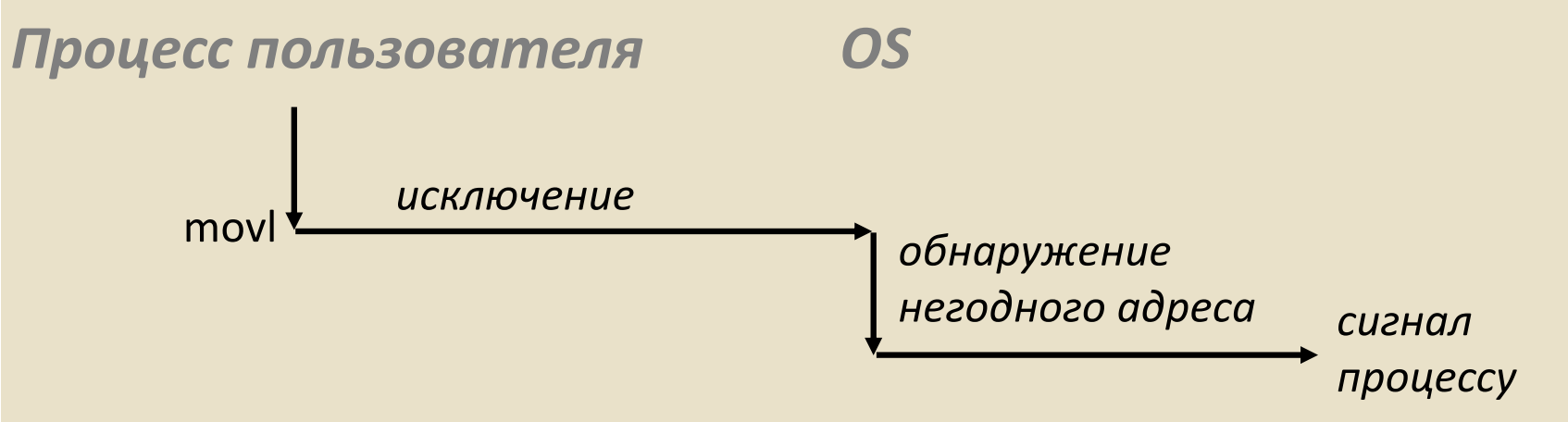


- Обработчик сбоя страницы загружает страницу в физическую память
- Возвращает управление на сбойную команду
- Сбойная команда успешно выполняется со второй попытки

Пример прекращения: негодный адрес (Invalid Memory Reference)

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:        c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



- Отправляет сигнал **SIGSEGV** процессу пользователя
- Процесс пользователя завершается с сообщением “segmentation fault”

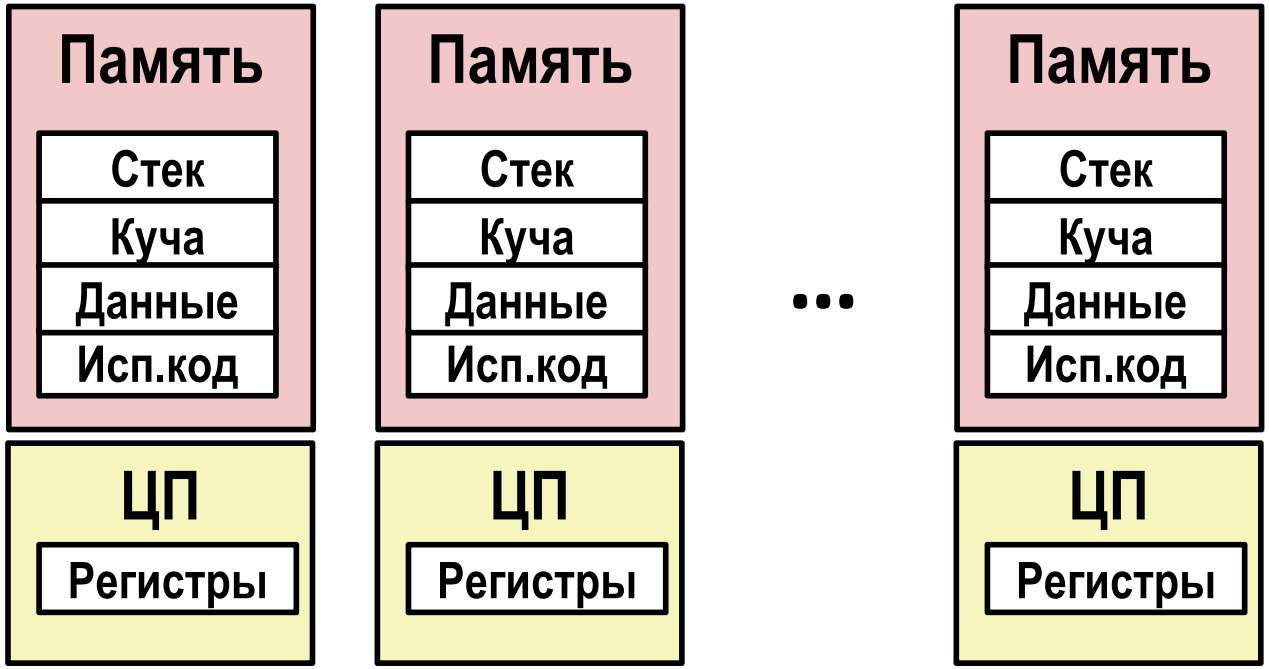
Исключения и процессы

- Поток управления с исключениями
- Исключения
- Процессы
- Управление процессами

Процессы

- **Определение: *процесс* – экземпляр исполнения программы**
  - Одна из фундаментальных концепций в информатике
  - Совсем не тоже самое, что “программа” или “процессор”
- **Процесс реализует каждой программе две ключевые абстракции (иллюзии) :**
  - ***Логический поток управления***
    - Каждой программе видится монопольное владение ЦП
    - Реализуется в ядре ОС механизмом *переключения контекста*
  - ***Частное виртуальное адресное пространство***
    - Каждой программе видится монопольное владение памятью
    - Реализуется в ядре ОС механизмом *виртуальной памяти*

Иллюзия многопроцессности



- **Компьютер исполняет несколько процессов одновременно**
  - Приложения для одного и более пользователей
    - Веб-обозреватели, почтовые клиенты, редакторы, …
  - Фоновые задачи
    - Наблюдение за сетью и прочими устройствами ввода-вывода

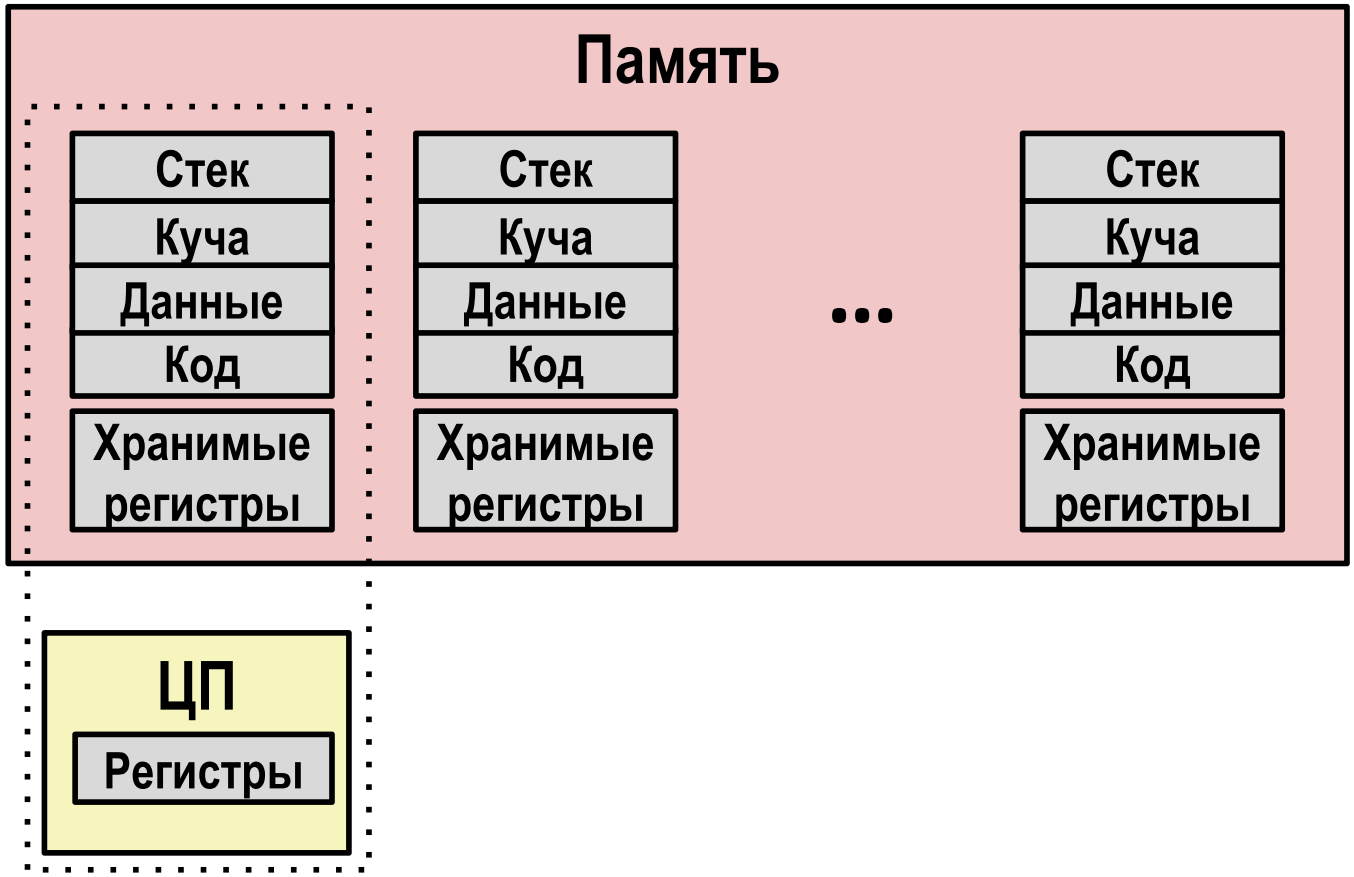
Пример многопроцессности

xterm

```
Processes: 123 total, 5 running, 9 stuck, 109 sleeping, 611 threads                11:47:07
Load Avg: 1.03, 1.13, 1.14  CPU usage: 3.27% user, 5.15% sys, 91.56% idle
SharedLibs: 576K resident, 0B data, 0B linkedit.
MemRegions: 27958 total, 1127M resident, 35M private, 494M shared.
PhysMem: 1039M wired, 1974M active, 1062M inactive, 4076M used, 18M free.
VM: 280G vsize, 1091M framework vsize, 23075213(1) pageins, 5843367(0) pageouts.
Networks: packets: 41046228/11G in, 66083096/77G out.
Disks: 17874391/349G read, 12847373/594G written.

PID    COMMAND      %CPU TIME     #TH   #WQ  #PORT #MREG RPRVT  RSHRD  RSIZE  VPRVT  VSIZE
99217- Microsoft Of 0.0  02:28.34 4     1    202   418   21M    24M    21M    66M    763M
99051  usbmuxd      0.0  00:04.10 3     1    47    66    436K   216K   480K   60M    2422M
99006  iTunesHelper 0.0  00:01.23 2     1    55    78    728K   3124K  1124K  43M    2429M
84286  bash         0.0  00:00.11 1     0    20    24    224K   732K   484K   17M    2378M
84285  xterm        0.0  00:00.83 1     0    32    73    656K   872K   692K   9728K  2382M
55939- Microsoft Ex 0.3  21:58.97 10    3    360   954   16M    65M    46M    114M   1057M
54751  sleep        0.0  00:00.00 1     0    17    20    92K    212K   360K   9632K  2370M
54739  launchdadd   0.0  00:00.00 2     1    33    50    488K   220K   1736K  48M    2409M
54737  top          6.5  00:02.53 1/1   0    30    29    1416K  216K   2124K  17M    2378M
54719  automountd   0.0  00:00.02 7     1    53    64    860K   216K   2184K  53M    2413M
54701  ocspd        0.0  00:00.05 4     1    61    54    1268K  2644K  3132K  50M    2426M
54661  Grab         0.6  00:02.75 6     3    222+  389+  15M+   26M+   40M+   75M+   2556M+
54659  cookied      0.0  00:00.15 2     1    40    61    3316K  224K   4088K  42M    2411M
53818  mdworker     0.0  00:01.67 4     1    52    91    7628K  7412K  16M    48M    2438M
50878  mdworker     0.0  00:11.17 3     1    53    91    [illegible]K  6148K  9976K  44M    2434M
[illegible]  [illegible]  [illegible]  [illegible]  [illegible]  [illegible]  [illegible]  [illegible]K  [illegible]K  532K   9700K  2382M
50078  emacs        0.0  00:06.70 1     0    20    35    52K    216K   88K    18M    2392M
```

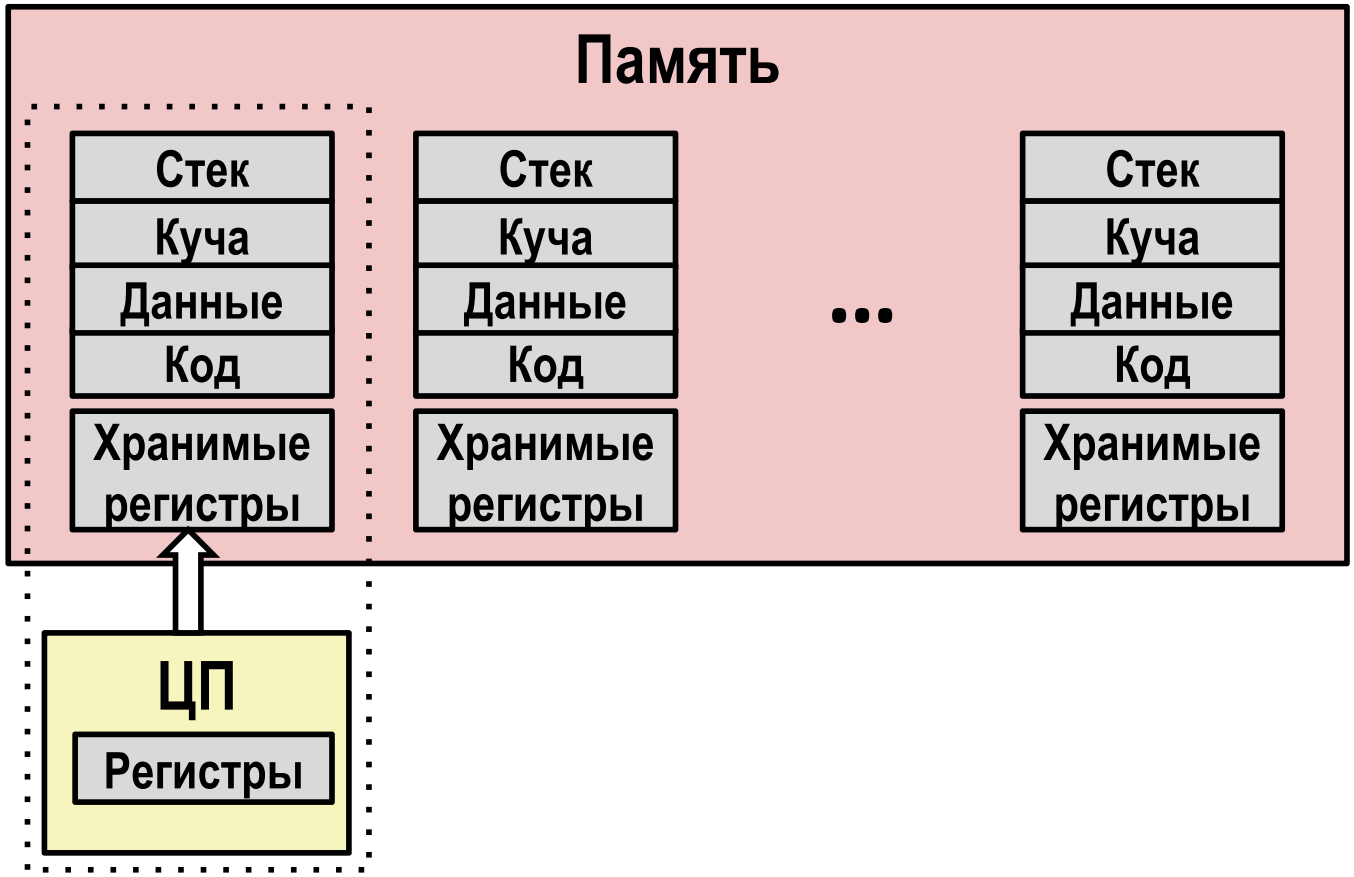
- **Исполнение программы “top” на Mac**
  - В системе 123 процесса, 5 из которых активны
  - Различаются по идентификатору процесса (PID)

Реальность многопроцессности (раньше)



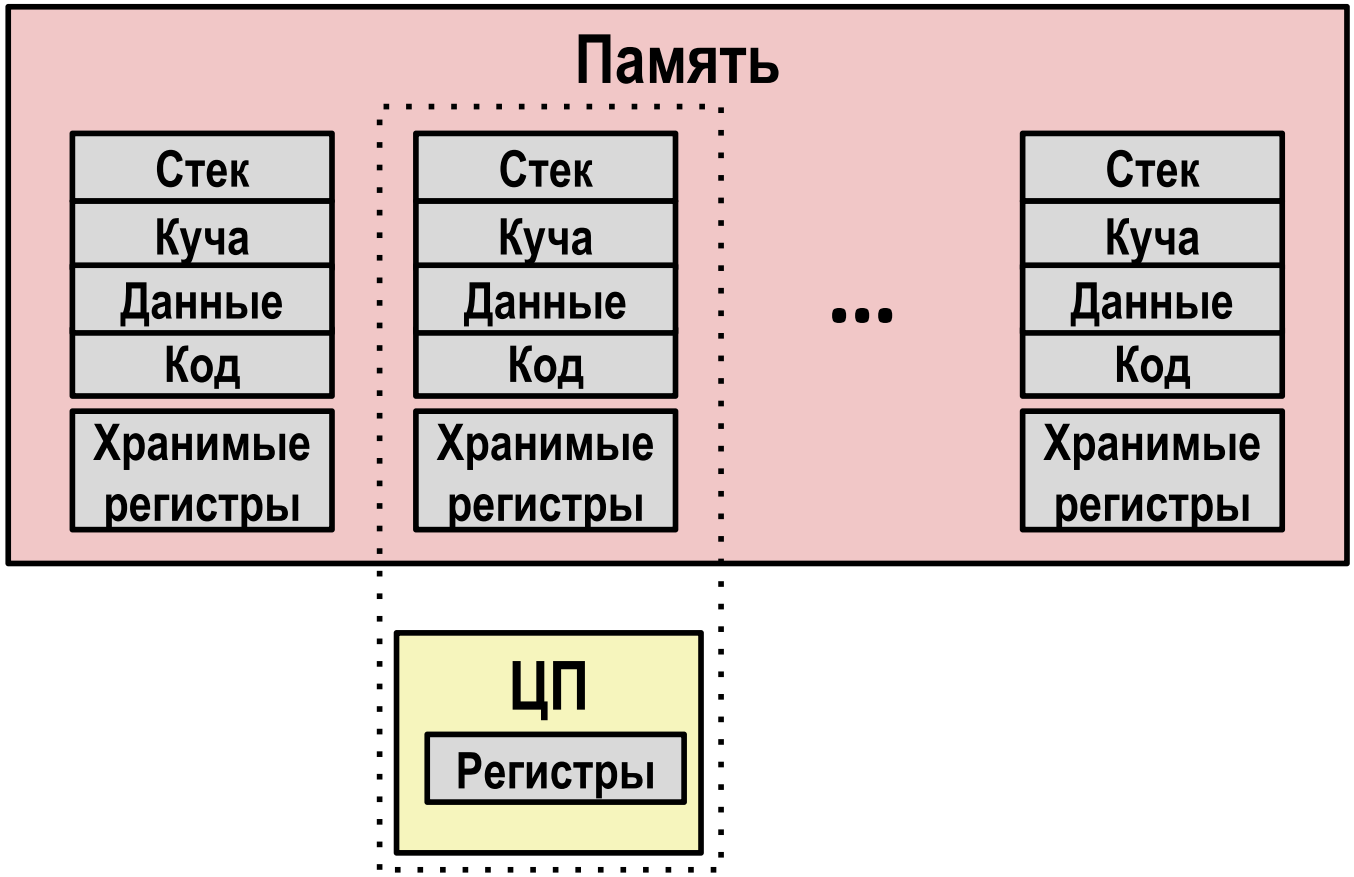
- **Один процессор исполняет несколько процессов параллельно**
  - Перекрывающиеся во времени выполнения процессов (многозадачность)
  - Адресные пространства управляются виртуальной памятью
  - Содержимое значений регистров неисполняемых процессов сохраняется в памяти

Реальность многопроцессности (раньше)



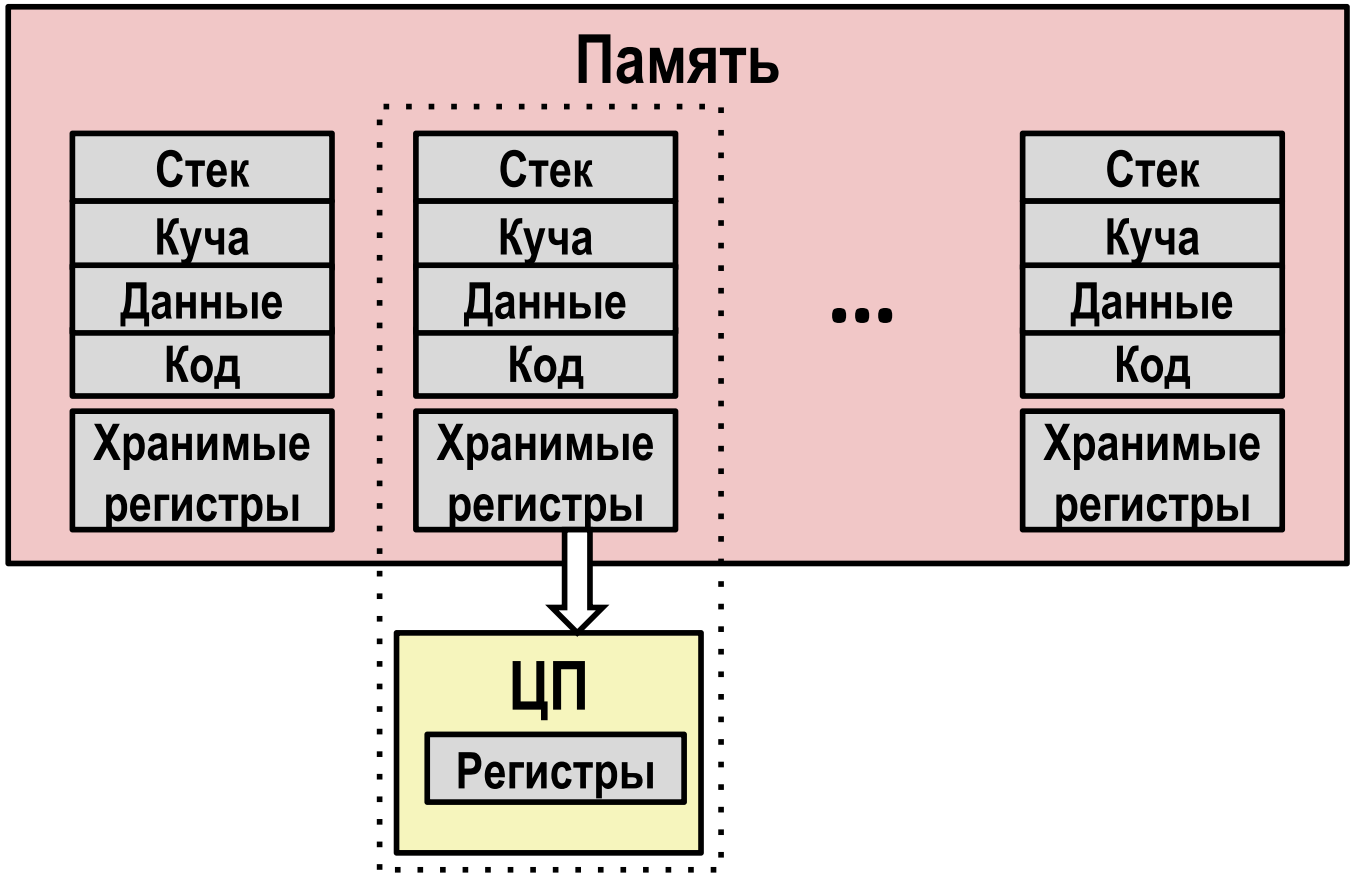
- **Текущее содержимое текущих регистров – в память**

Реальность многопроцессности (раньше)



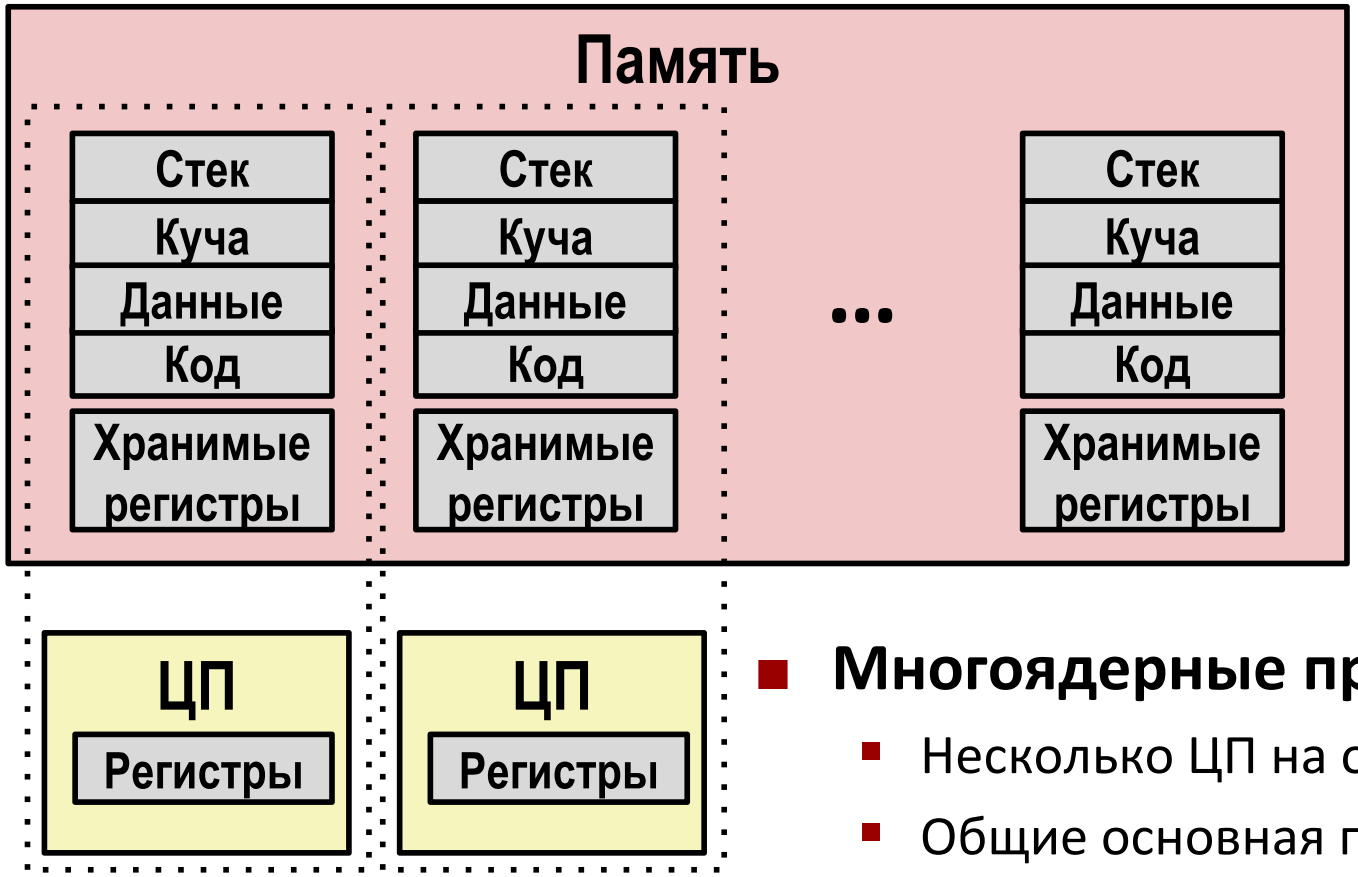
- **Выбрать следующий процесс для выполнения**

Реальность многопроцессности (раньше)



- **Загрузить сохранённые регистры и переключить адресное пространство (переключить контекст)**

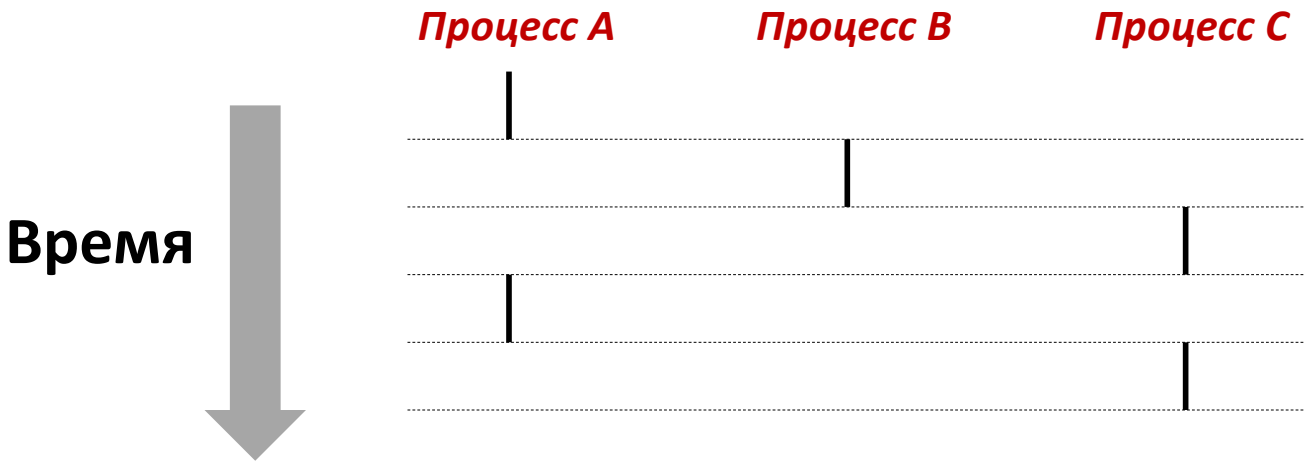
Реальность многопроцессности (сейчас)



- **Многоядерные процессоры**
  - Несколько ЦП на одном кристалле
  - Общие основная память и некоторые кеши
  - Каждый исполняет свой процесс
    - ядро ОС распределяет процессы по ядрам кристалла

Одновременные процессы

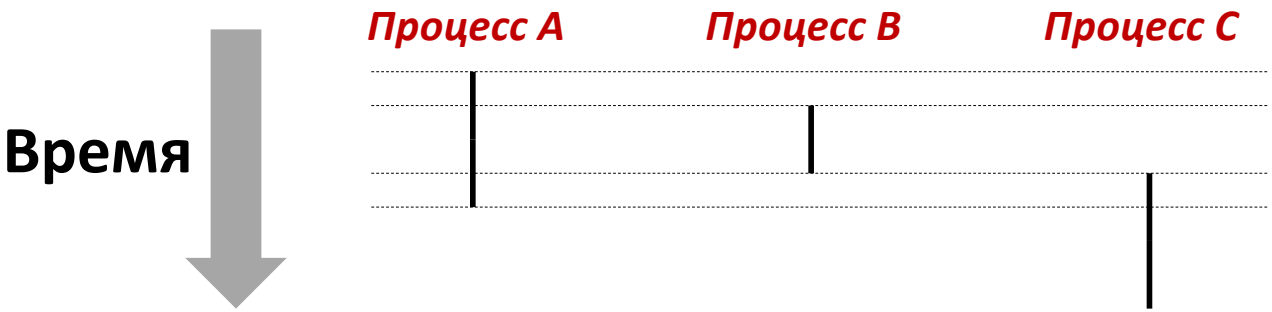
- **Два процесса идут *одновременно* (*являются одновременными)* если каждый из процессов начинается до завершения другого**
- **В противном случае, они являются *последовательными***
- **Примеры (исполнение на единственном ядре ЦП):**
  - Одновременные: A & B, A & C
  - Последовательные: B & C



Точка зрения пользователя на одновременные процессы

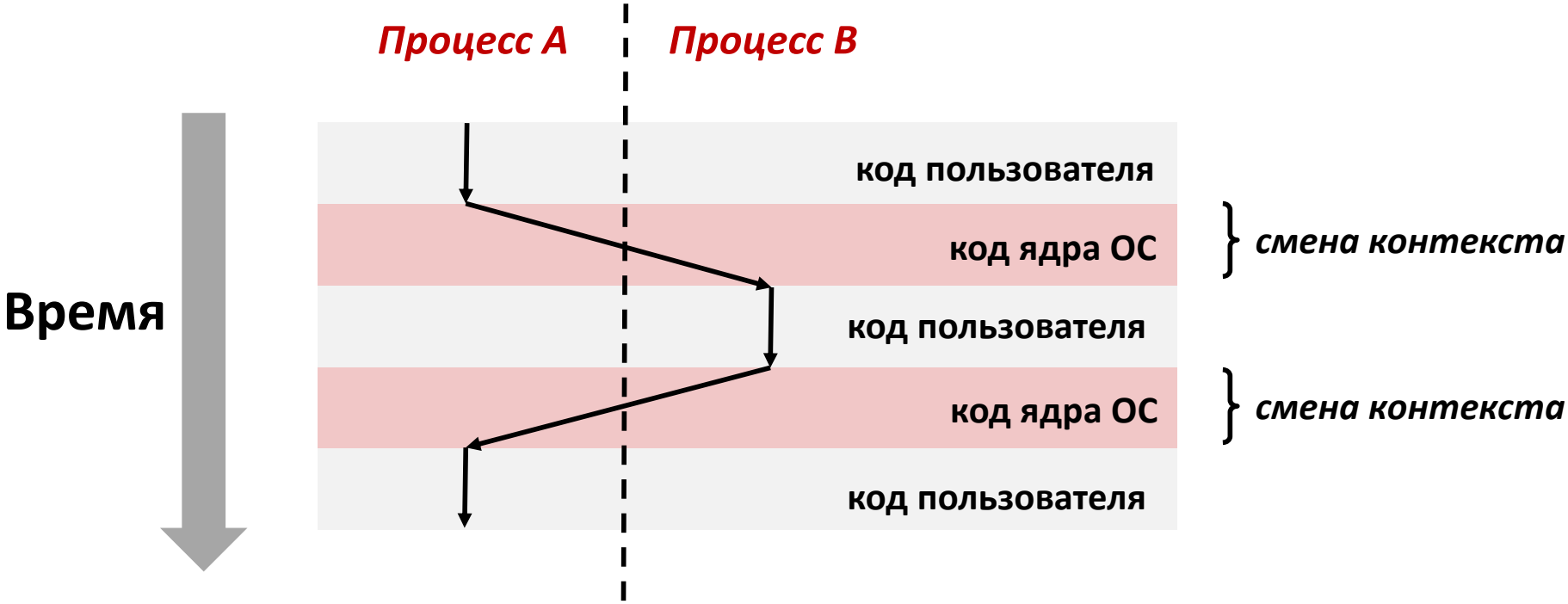
- Потоки управления одновременных процессов физически разнесены во времени

- При этом, мы можем мыслить одновременные процессы идущими параллельно друг с другом



Смена контекста процессов

- **Процессы управляются частью исполняемого кода ОС, под названием *ядро ОС (kernel)***
  - Важно: ядро ОС не отдельный процесс, но выполняется в паузах некоторых пользовательских процессов
- **Физический поток управления переходит от одного процесса к другому посредством *смены контекста***



Исключения и процессы

- Поток управления с исключениями
- Исключения
- Процессы
- Управление процессами

Обработка ошибок вызова системы

- **Если ошибка, то функции системного уровня Unix обычно возвращают -1 и устанавливают глобальную переменную `errno` для обозначения причины.**
- **Тяжелое но быстрое правило:**
  - Вы обязаны проверять результат каждого вызова функций системного уровня
  - Исключение – немногие функции, типа `void`

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(0);
}
```

Функции сообщений об ошибках

- **Проще использовать *функции сообщений об ошибках*:**

```
void unix_error(char *msg) /* Ошибки в стиле Unix */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(0);
}
```

```
 if ((pid = fork()) < 0)
    unix_error("fork error");
```

Обертки обработки ошибок

- **Ещё упростим демонстрируемый код обёртками в другом стиле:**

```
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

```
  pid = Fork();
```

Запрос идентификатора процесса (PID)

- `pid_t getpid(void)`
  - Возвращает PID текущего процесса
- `pid_t getppid(void)`
  - Возвращает PID родительского процесса

Создание и завершение процессов

Программист мыслит о 3 возможных состояниях процесса

- **Исполняется**
  - Процесс или исполняется, или ожидает исполнения и будет выбран ядром ОС для исполнения
- **Остановлен**
  - Исполнение процесса отложено, и не будет возобновляться до специального сигнала
- **Завершён**
  - Процесс необратимо прекращён

Завершение процесса

- **Процесс завершается по 1 из 3 причин:**
  - Получает сигнал, по которому должен завершиться
  - Возвращает управление из программы `main`
  - Вызывает функцию `exit`

- **`void exit(int status)`**
  - Завершает процесс с кодом завершения `status`
  - Соглашение: код нормального завершения 0, не 0 – при ошибке
  - Другой способ явно указать код завершения вернуть целое значение из программы `main`

- **`exit` вызывается однажды и никогда не возвращает**

Создание процессов

- ***Родительский процесс*** **создаёт новый** ***дочерний процесс*** **обращением к `fork`**
- **`int fork(void)`**
  - Возвращает 0 дочернему процессу, и PID дочернего - родительскому
  - Дочерний *почти* идентичен родительскому:
    - Дочерний получает идентичную (но отдельную) копию родительского виртуального адресного пространства.
    - Дочерний получает идентичные копии описаний файлов открытых родительским
    - Дочерний получает PID, иной чем у родителя

- **`fork` привлекает (и часто запутывает) потому, что вызыванный *однажды* возвращает *дважды***

Пример вызова `fork`

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Потомок */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Родитель */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

```
linux> ./fork
parent: x=0
child : x=2
```

- **На 1 вызов, 2 возврата**
- **Одновременная работа**
  - **Нельзя предсказать порядок выполнения родителя и потомка**
- **Отдельная копия адресного пространства**
  - **`x` равен 1, когда fork возвращает и родителю и потомку**
  - **последующие изменения `x` независимы**
- **Вместе используют файлы**
  - **`stdout` одинаков и в родителе и в потомке**

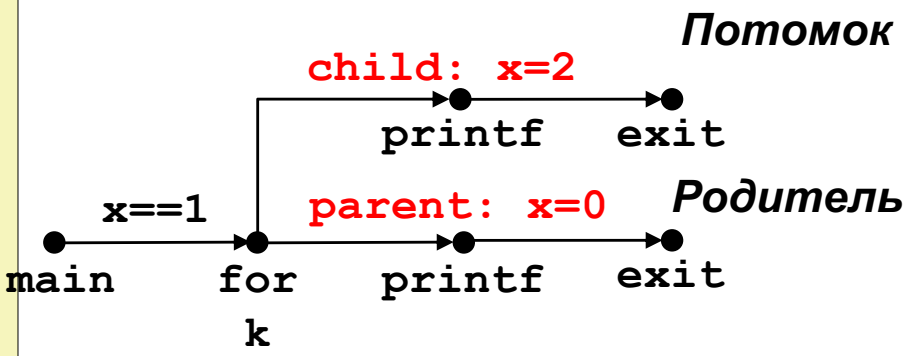
Моделирование `fork` графом процессов

- ***Граф процессов* – удобное средство понимания честивной упорядоченности операторов в параллельной программе:**
  - Каждая вершина – исполнение оператора
  - a -> b означает, что `a` произошло раньше b
  - Ребра помечаются текущими значениями переменных
  - Вершины `printf` помечаются выводимым текстом
  - Каждый граф начинается вершиной без входящих рёбер
- **Любая *топологическая сортировка* графа соответствует возможному полному порядку.**
  - Полный порядок вершин, при котором все ребра указывают слева направо

Пример графа процессов

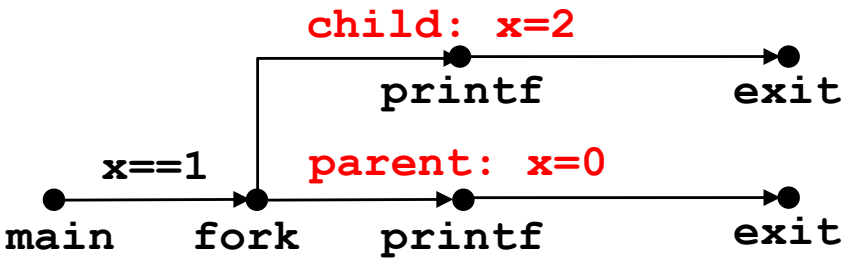
```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Потомок */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Родитель */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

fork.c

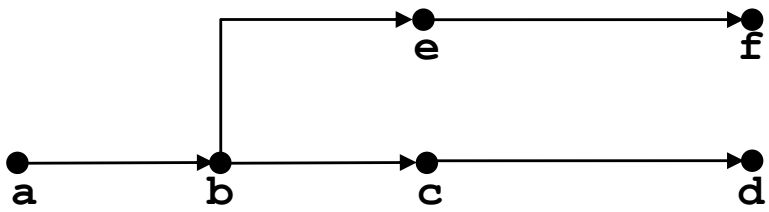


Интерпретация графа процессов

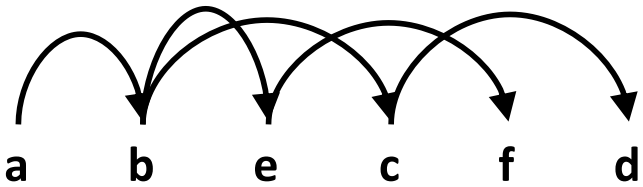
- **Исходный граф:**



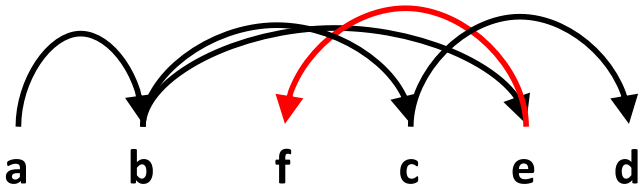
- **Переразмеченный graph:**



Возможный порядок:



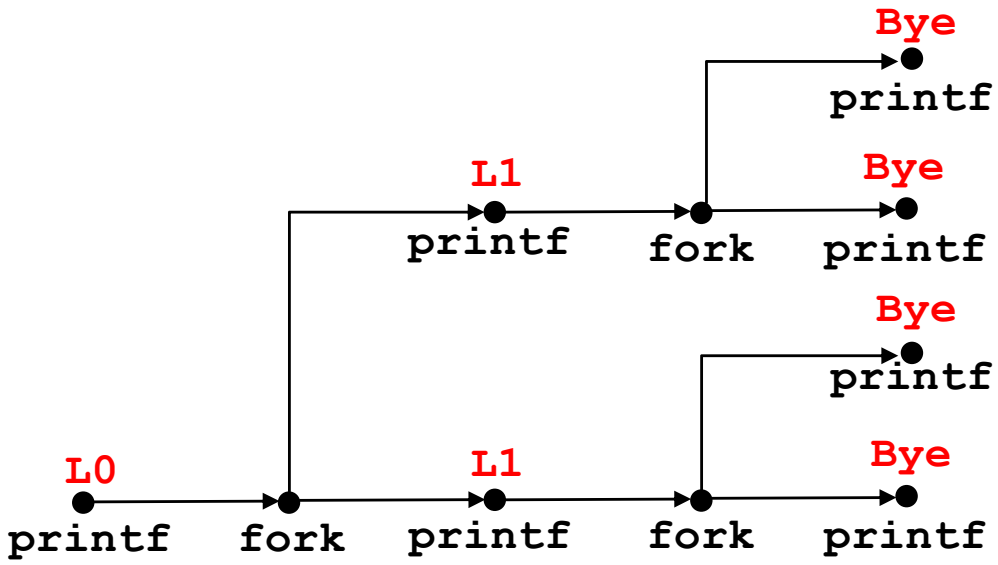
Невозможный порядок:



2 последовательных fork-а

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

forks.c



Возможный вывод:

L0
L1
Bye
Bye
L1
Bye
Bye

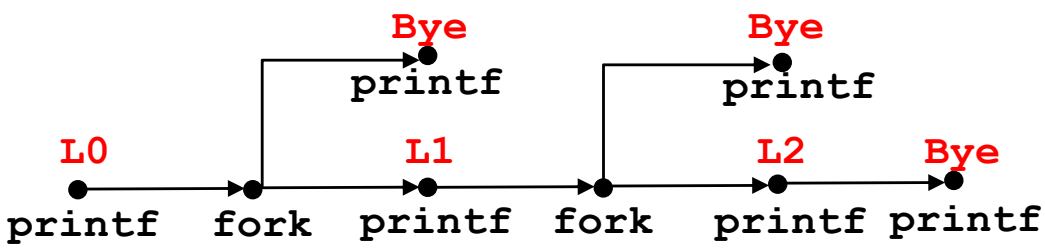
Невозможный вывод:

L0
Bye
L1
Bye
L1
Bye
Bye

Вложенные fork-и в родителе

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c



Возможный вывод:

L0
L1
Bye
Bye
L2
Bye

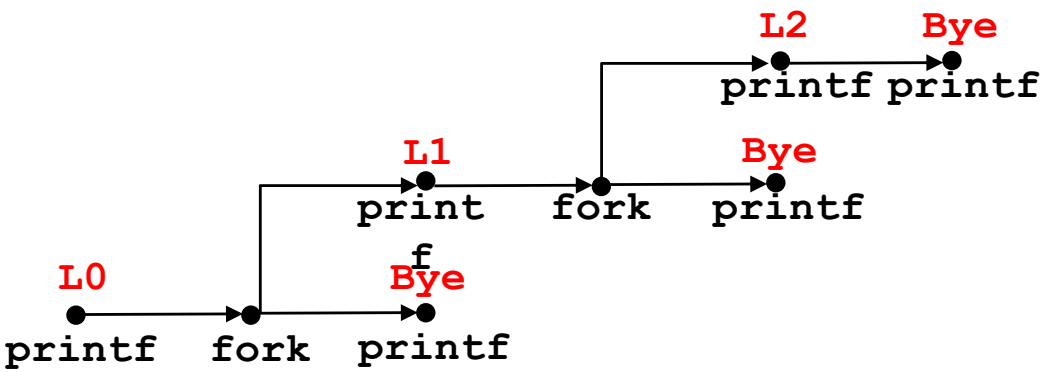
Невозможный вывод:

L0
Bye
L1
Bye
Bye
L2

Вложенные fork-и в потомке

```c
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c



Возможный вывод:

L0
Bye
L1
L2
Bye
Bye

Невозможный вывод:

L0
Bye
L1
Bye
Bye
L2

Скашивание дочерних процессов

- **Идея**
  - Когда процесс завершён, он всё ещё потребляет ресурсы ОС
    - Различные системные таблицы
  - Называется “зомби”
    - Живой труп, полуживой полумёртвый
- **Скашивание (reaping)**
  - Выполняется родительским процессом над завершённым дочерним
  - Родительский процесс получает статус завершения дочернего
  - Ядро ОС очищает системные таблицы от данных дочернего процесса
- **Что если родительский процесс не скосит дочерний?**
  - если родительский завершится не срезав дочерние, то дочерние будут срезаны процессом–прародителем **`init`**
  - явное срезание необходимо долгоживущим процессам
    - например оболочкам и серверам

Зомби

```c
void fork7() {
    if (fork() == 0) {
        /* Потомок */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Бесконечный цикл */
    }
}
```

forks.c

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- **ps** показывает потомка “defunct” (например, зомби)
- Кончина родителя позволяет **init**-у скосить потомка

Бесконечный потомок

```c
void fork8()
{
    if (fork() == 0) {
        /* Потомок */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Бесконечный цикл */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

forks.c

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- Потомок активен, несмотря на кончину родителя
- Необходимо кончить потомка явно, или он будет выполняться без конца.

`wait`: синхронизация с потомком

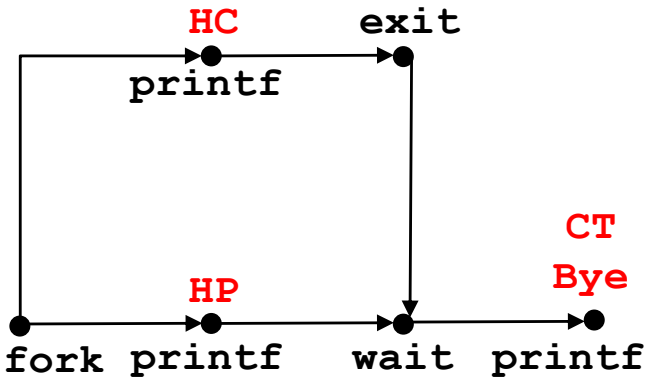
- **Родитель скашивает потомка вызовом функции `wait`**

- **`int wait(int *child_status)`**
  - Останавливает текущий процесс до кончины одного из потомков
  - Возвращает **`pid`** законченного потомка
  - Если **`child_status != NULL`**, то он указывает на значение причины и статус завершения:
    - Проверять можно макросами из `wait.h`
      - `WIFEXITED, WEXITSTATIS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`
      - Детали в книге

wait: синхронизация с потомком

```c
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

forks.c



Возможный вывод:

HC
HP
CT
Bye

Невозможный вывод:

HP
CT
Bye
HC

Ещё wait

- Несколько потомков кончаются в произвольном порядке
- Для получения информации о статусах завершения полезны макросы WIFEXITED и WEXITSTATUS

```
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Потомок */
        }
    for (i = 0; i < N; i++) { /* Родитель */
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

forks.c

waitpid: ожидание определённого процесса

- **pid_t waitpid(pid_t pid, int &status, int options)**
  - Останавливает текущий процесс до кончины указанного
  - Варианты смотрите в книге

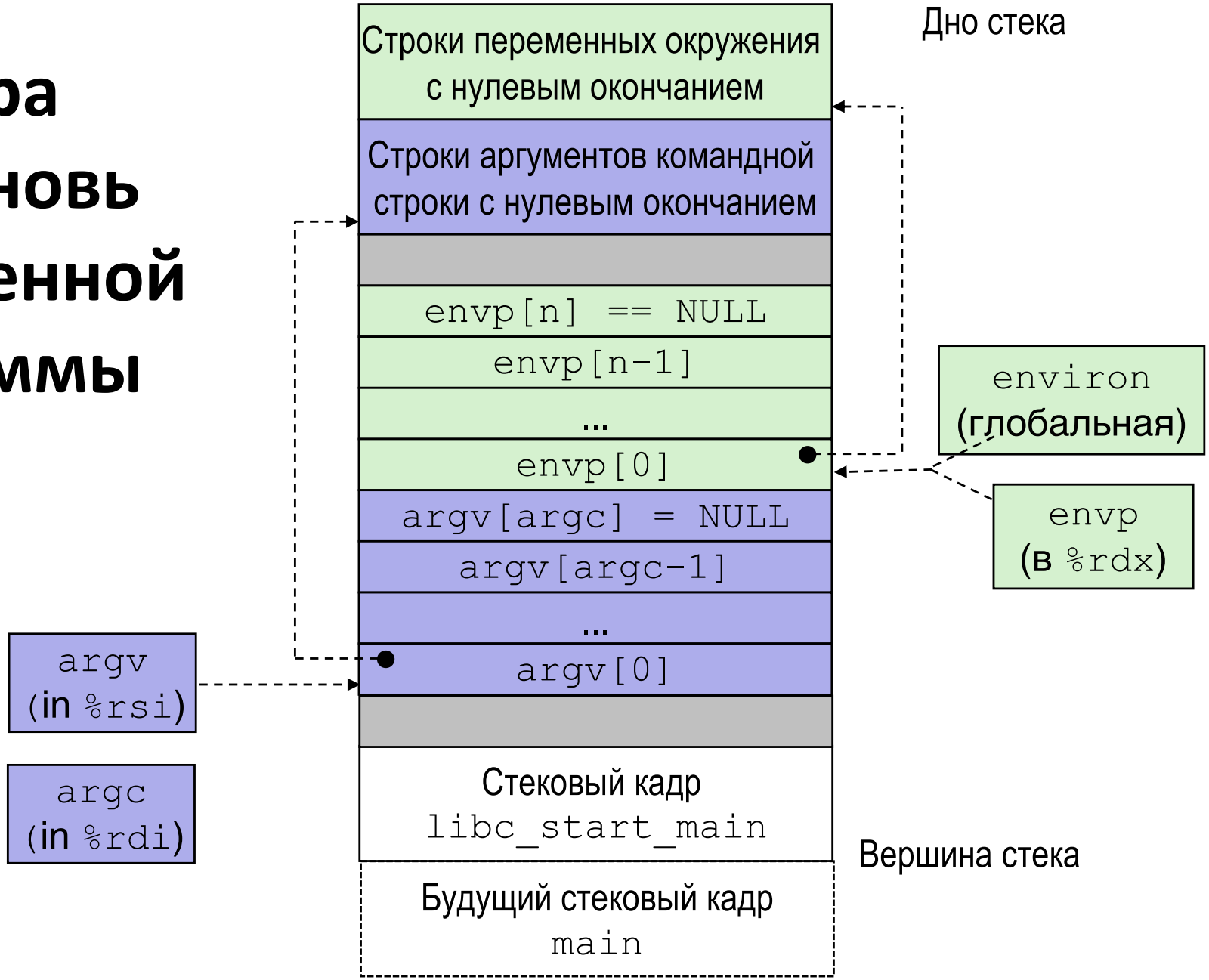
```
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

forks.c

`execve`: загрузка и запуск программ

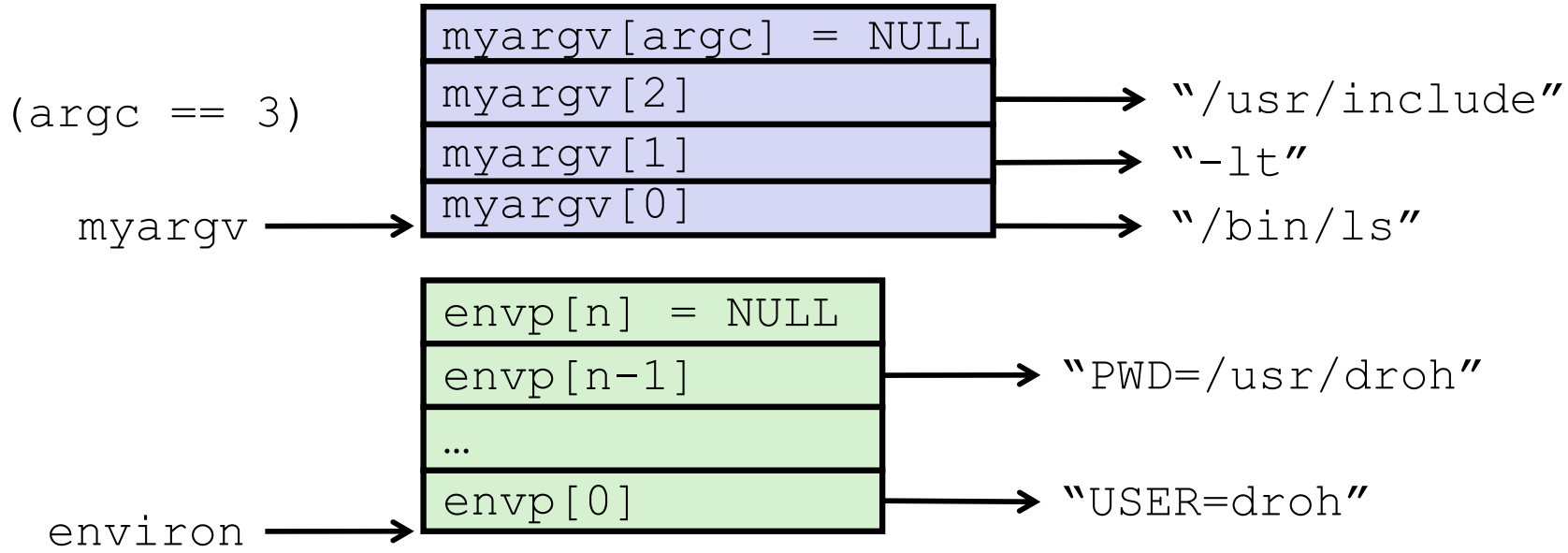
- **`int execve(char *filename, char *argv[], char *envp[])`**
- **Загружает и запускает в текущем процессе:**
  - Исполняемый файл **`filename`**
    - Может быть загружаемым файлом или сценарием начинающимся с `#!interpreter` (например, `#!/bin/bash`)
  - …со списком аргументов **`argv`**
    - По соглашению **`argv[0]==filename`**
  - …and списком переменных окружения **`envp`**
    - Строки “имя=значений” (например, `USER=droh`)
    - `getenv, putenv, printenv`
- **Перезаписывает code, data и stack**
  - Сохраняет PID, открытые файлы и контекст сигналов context
- **Однажды вызванная никогда не возвращает, т.к. некуда**
  - …кроме ошибочных ситуаций

Структура стека вновь загруженной программы



Пример execve

- **Выполняет** `"/bin/ls -lt /usr/include"` **в дочернем процессе используя текущее окружение:**




```
if ((pid = Fork()) == 0) {   /* Потомок загружает программу */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Сводка

- **Исключения**
  - События, требующие процессов, отличных от процесса программы
  - Возникают вовне (прерывания) или внутри (ловушки и сбои)

- **Процессы**
  - В любой момент, в системе активны несколько процессов
  - Только один может исполняться на одном ядре
  - Каждому процессу кажется, что он полностью контролирует процессор и частное пространство памяти

Сводка (продложение)

- **Размножение процессов**
  - Вызов `fork`
  - На каждый вызов два возврата
- **Завершение процесса**
  - Вызов `exit`
  - После вызова нет возврата
- **Ожидание завершения процессов и скашивание**
  - Вызовы `wait` или `waitpid`
- **Загрузка и исполнение программ**
  - Вызов `execve` (или аналога)
  - После вызова (в норме) нет возврата