Поток управления с исключениями

Основы информатики

Компьютерные основы программирования

u.to/DbCmFA

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

u.to/XoKmFA

Лекция 12, 28 апреля 2023

Лектор:

Дмитрий Северов, кафедра информатики 608 КПМ

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

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:**
  - Переходы - безусловный и условные
  - 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”**

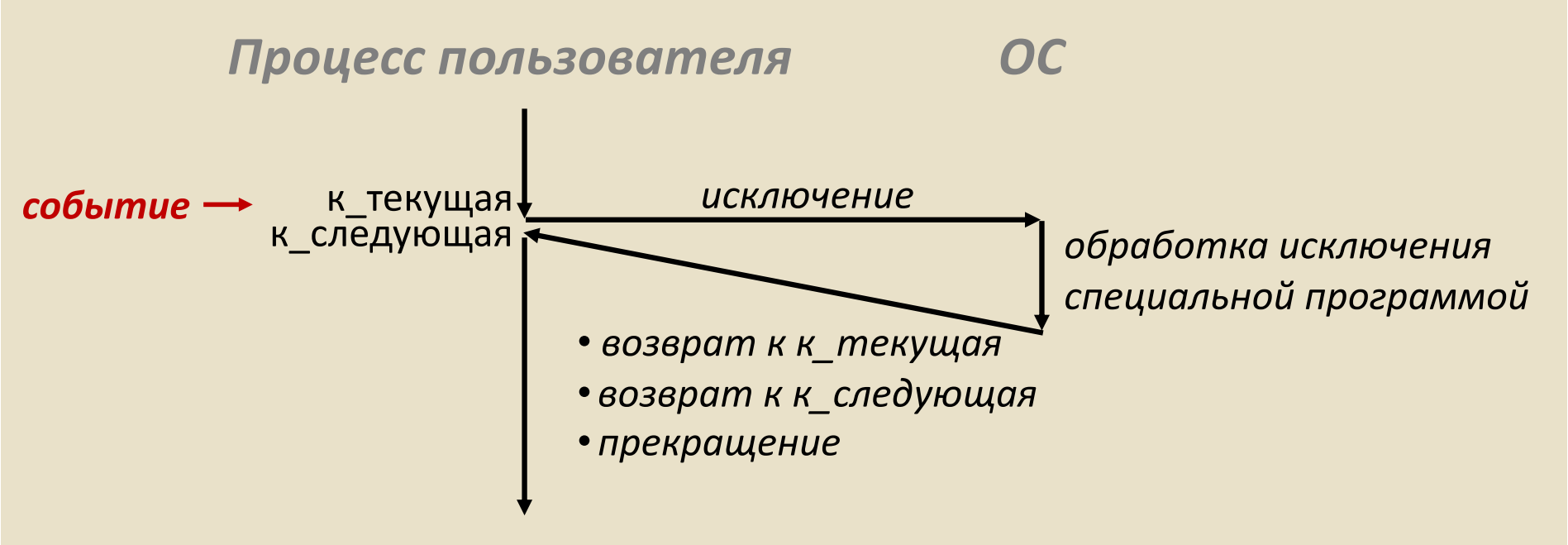
Поток управления с исключительными ситуациями (исключениями)

- **Присутствует на всех уровнях компьютерной системы**

- **Низкоуровневые механизмы**
  - 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**
  - Смена процессных контекстов
  - Сигналы
  - Нелокальные переходы : setjmp()/longjmp()
  - Реализуются средой исполнения:
    - ПО операционной системы (смена контекстов и сигналы)
    - библиотеки языка С (нелокальные переходы)

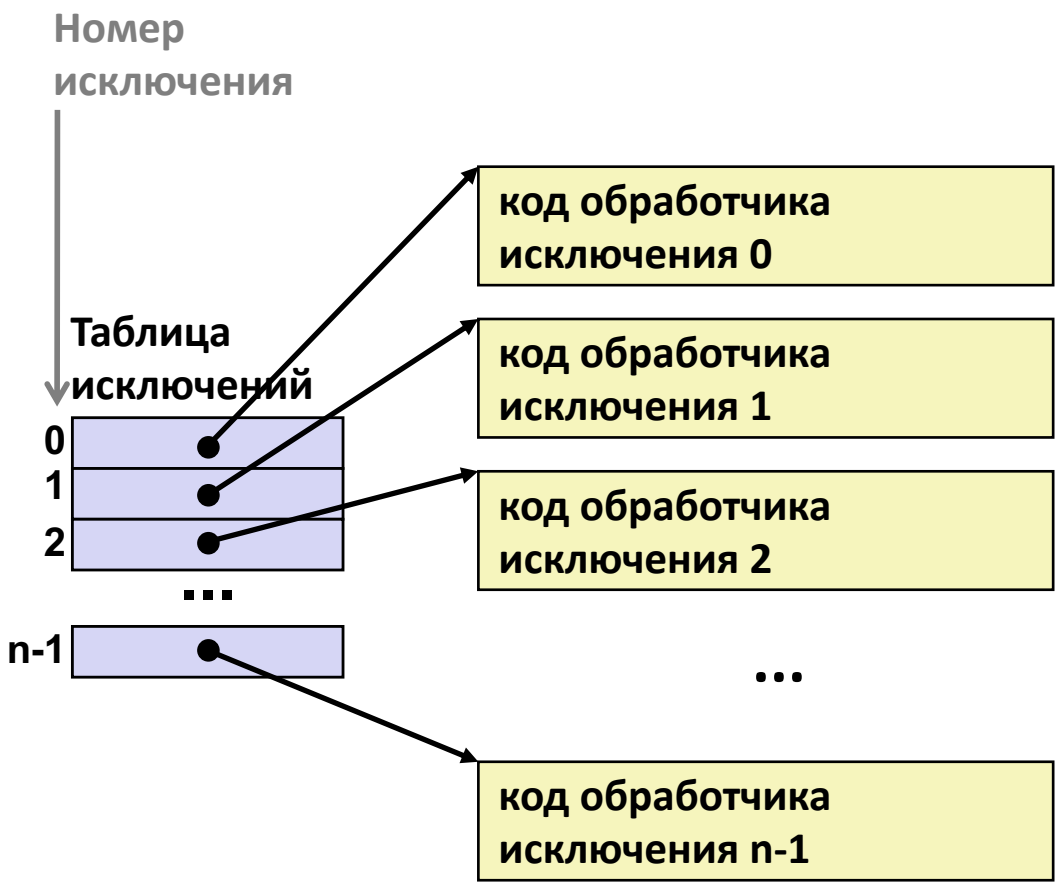
Исключения (исключительные ситуации)

- ***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:**
  - Ввод/вывод
    - нажатие Ctrl-C на клавиатуре
    - поступление пакета данных из сети
    - поступление блока данных с диска
  - Жёсткая перезагрузка (система в начальном состоянии)
    - нажатие кнопки «reset»
  - Мягкая перезагрузка (часть ПО в исходном состоянии)
    - нажатие Ctrl-Alt-Delete на ПК

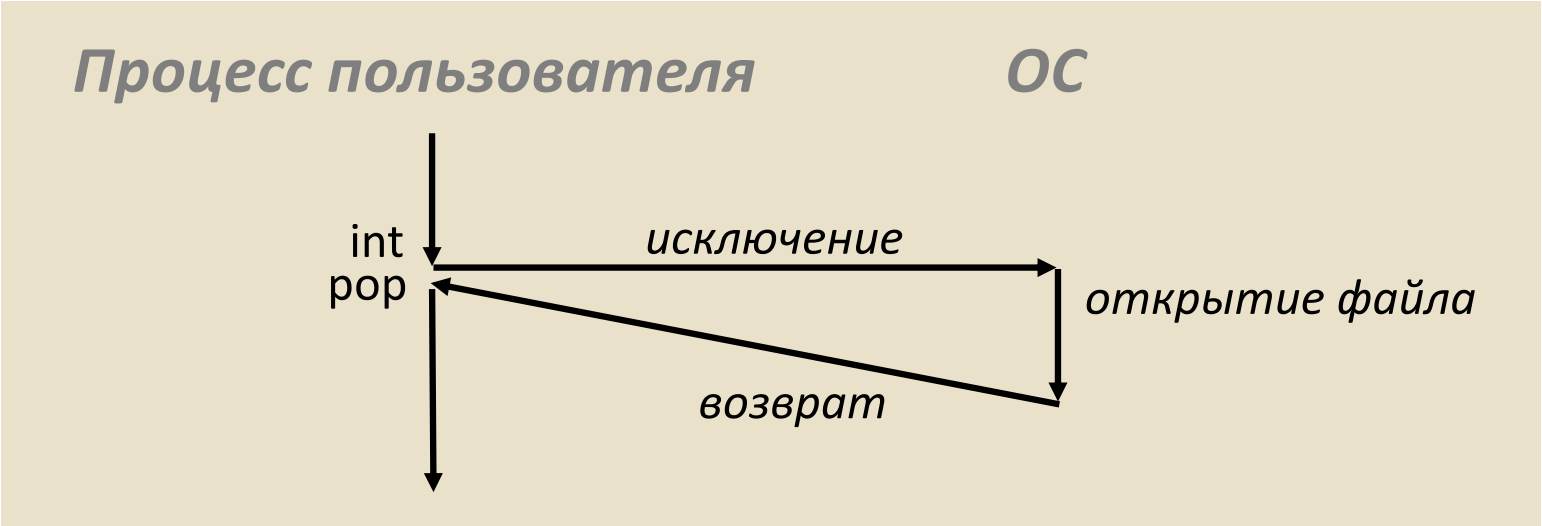
Синхронные исключения

- **Возникают в результате исполнения команд:**
  - ***Ловушки (Traps)***
    - преднамеренные
    - примеры: ***обращения к ядру ОС***, точки останова, специальные команды
    - 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
    - прекращает текущую программу

Пример ловушки: открытие файла

- Пользователь вызывает: **`open(filename, options)`**
- Функция **`open`** выполняет команду системного вызова **`int`**

```
0804d070 <__libc_open>:
 . . .
 804d082:      cd 80                    int    $0x80
 804d084:      5b                       pop    %ebx
 . . .
```



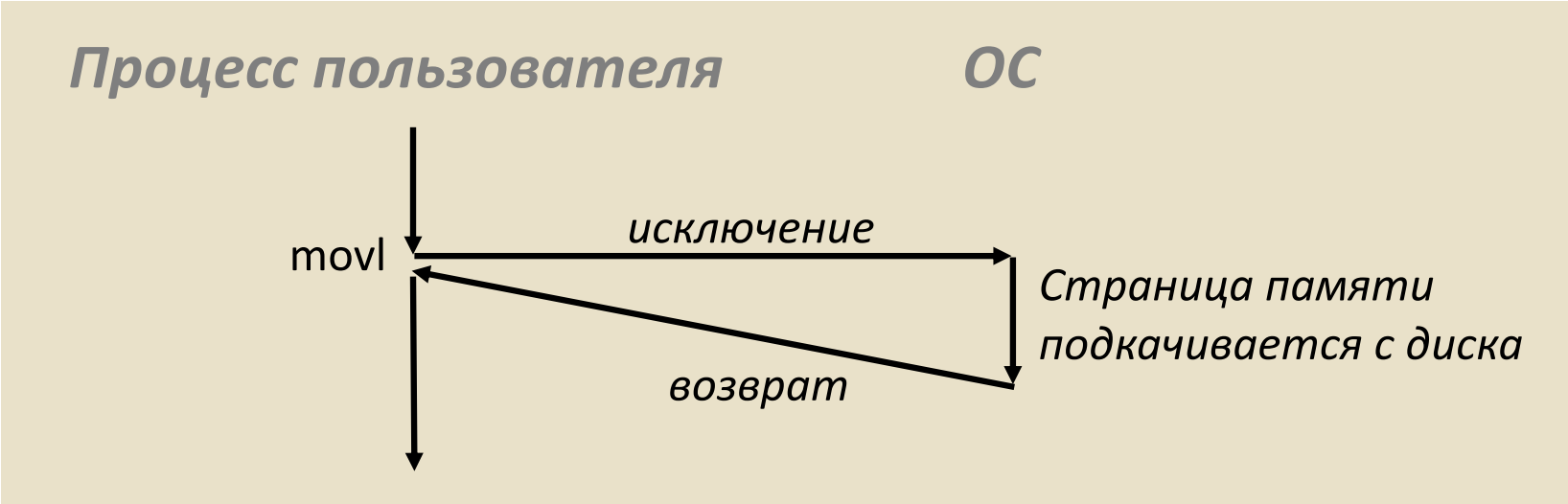
- ОС находит или создаёт файл и готовит его к чтению или записи
- Возвращает целочисленный дескриптор файла

Пример сбоя: 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
```

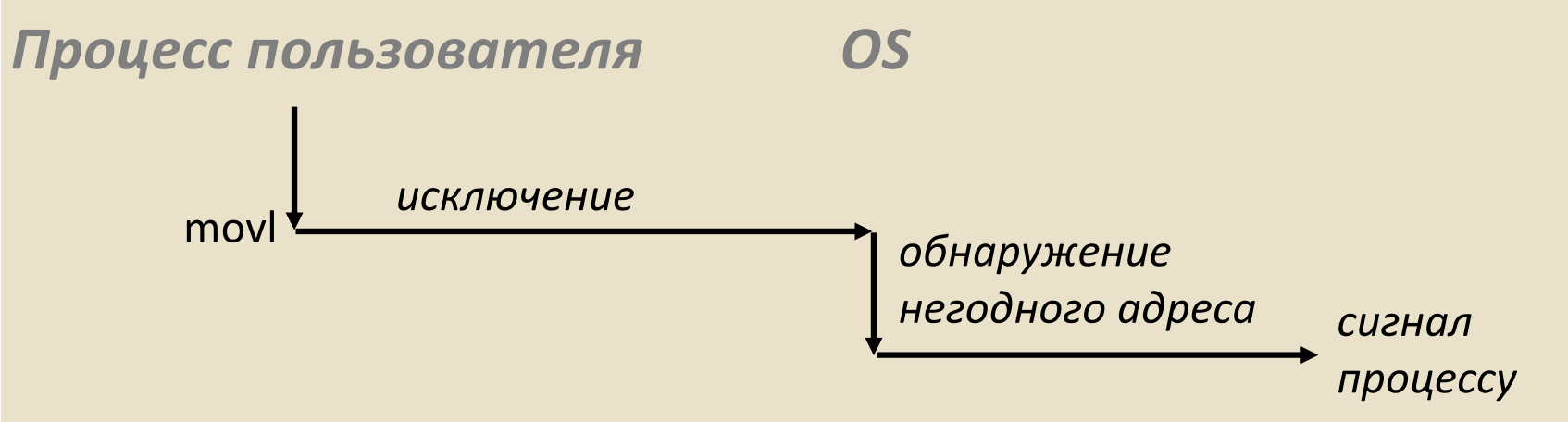


- Обработчик сбоя страницы загружает страницу в физическую память
- Возвращает управление на сбойную команду
- Сбойная команда успешно выполняется со второй попытки

Пример сбоя: негодный адрес
(Invalid Memory Reference)

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

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



- Обработчик сбоя страницы обнаруживает негодный адрес
- Отправляет сигнал **SIGSEGV** процессу пользователя
- Процесс пользователя завершается с сообщением “segmentation fault”

Таблица исключений (частично)

Номер исключения	*Описание*	*Класс исключения*
0	Divide error	Fault
13	General protection	Fault
14	Page fault	Fault
16	Floating-Point Error	Fault
18	Machine check	Abort
19	SIMD Floating-Point Exception	Fault
32-127	Maskable Interrupts	Interrupt or trap

Процессы

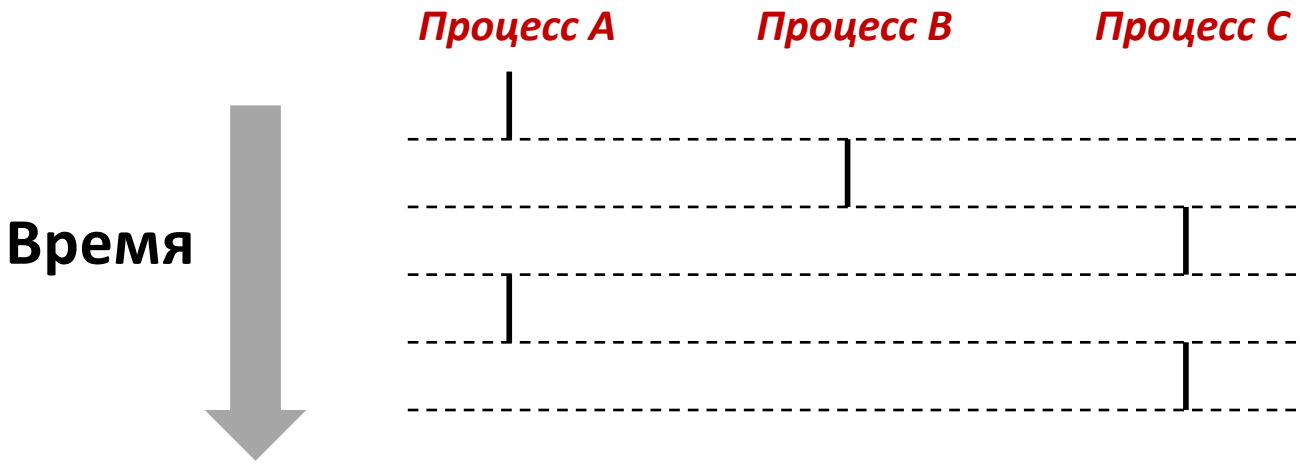
- **Определение: *процесс* – экземпляр исполнения программы**
  - Одна из фундаментальных концепций в информатике
  - Совсем не тоже самое, что “программа” или “процессор”

- **Процесс реализует каждой программе две ключевые абстракции (иллюзии) :**
  - Логический поток управления:
    - Каждой программе видится монопольное владение ЦП
  - Частное виртуальное адресное пространство
    - Каждой программе видится монопольное владение памятью

- **Чем обеспечиваются такие иллюзии?**
  - Исполнение процессов перекрывается во времени (многозадачность) или происходит на различных ядрах ЦП (cores)
  - Адресное пространство обеспечивает подсистема виртуальной памяти

Одновременные процессы

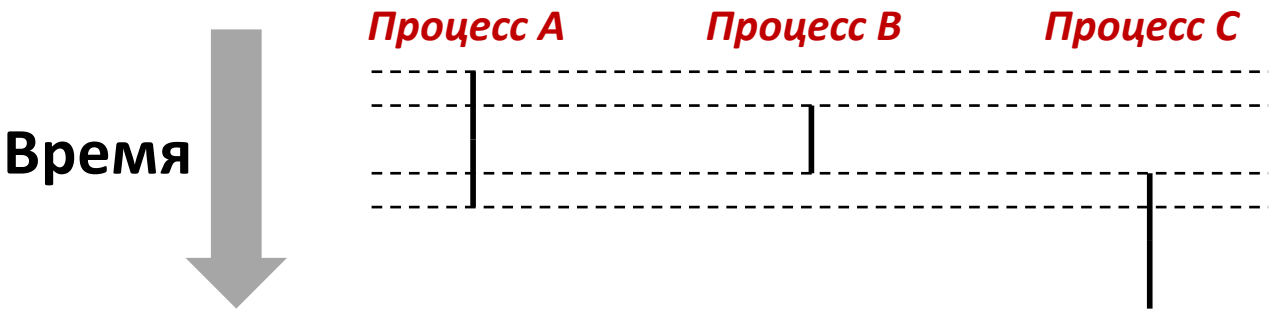
- **Два процесса идут *одновременно* (*являются одновременными)* если каждый из процессов начинается до завершения другого**
- **В противном случае, они являются *последовательными***
- **Примеры (исполнение на единственном ядре):**
  - Одновременные: A & B, A & C
  - Последовательные: B & C



Точка зрения пользователя на одновременные процессы

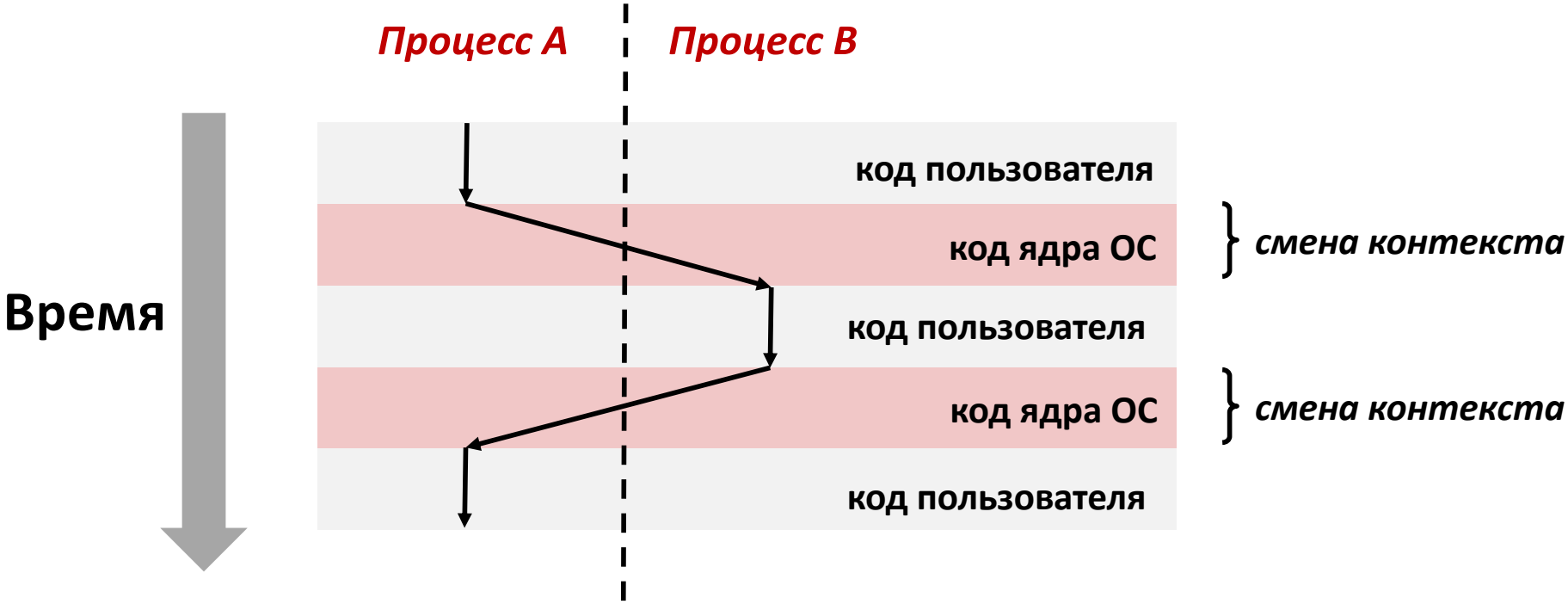
- Потоки управления одновременных процессов физически разнесены во времени

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



Смена контекста

- **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***



`fork`: создание нового процесса

- `int fork(void)`
  - порождает новый процесс (дочерний) идентичный вызвавшему процессу (родительскому)
  - возвращает 0 дочернему процессу
  - возвращает идентификатор дочернего процесса родительскому

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

- **`fork()` интересен (и сложен для восприятия) тем, что вызывается *однажды,* а возвращает управление *дважды***

Попробуем понять fork

Родительский процесс n

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Дочерний процесс m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`hello from parent` *Какой появится раньше?* `hello from child`

Fork: пример №1

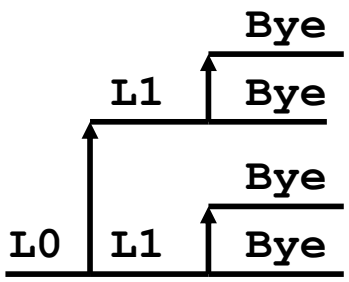
- **Родительский и дочерний исполняют один код**
  - Различаются результатом исполнения **`fork`**
- **Стартуют из одинаковых состояний, но каждый из своего**
  - В том числе разделяемый дескриптор стандартного потока вывода
  - Взаимное упорядочение исполнений функции вывода не определено

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork: пример №2

- **Родительский и дочерний могут продолжать плодиться**

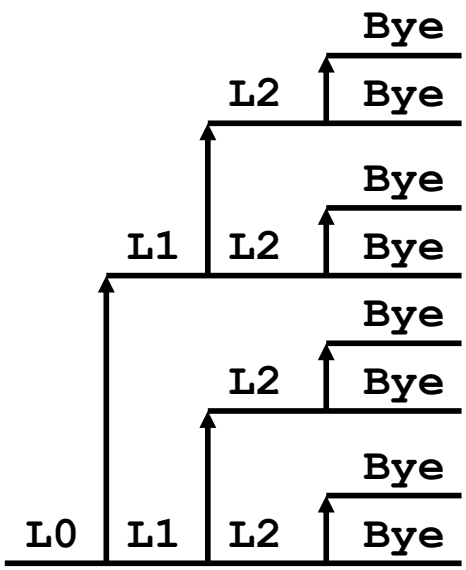
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork: пример №3

- **Родительский и дочерний могут продолжать плодиться**

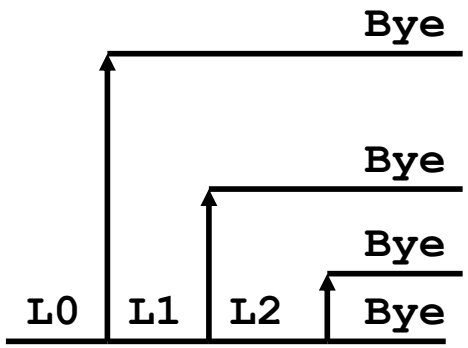
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork: пример №4

- **Родительский и дочерний могут продолжать плодиться**

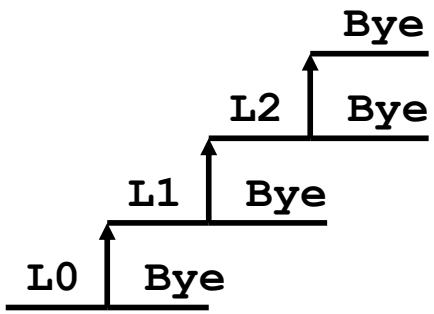
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



Fork: пример №5

- **Родительский и дочерний могут продолжать плодиться**

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



`exit`: окончание процесса

- **`void exit(int status)`**
  - выход из процесса
    - При нормальном завершении возвращает status 0
  - **`atexit()`** регистрирует функцию, выполняемую при завершении процесса

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Зомби

- **Идея**
  - Когда процесс завершён, он всё ещё потребляет ресурсы ОС
    - Различные системные таблицы
  - Называется “зомби”
    - Живой труп, полуживой полумёртвый
- **Срезание (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 срежет дочерний?**
  - если родительский завершится не срезав дочерние, то дочерние будут срезаны процессом–прародителем **`init`**
  - явное срезание необходимо долгоживущим процессам
    - например оболочкам и серверам

Зомби, пример

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

```
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 дочерний процесс как “defunct”
- Убиение родительского позволит прародителю **init** срезать дочерний

Безостановочный дочерний,пример

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

```
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
```

- Дочерний процесс активен несмотря на убиение родительского
- Должен быть убит явно, или останется исполняться бесконечно.

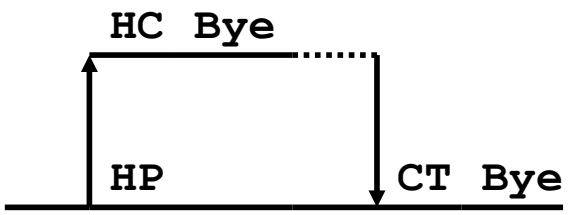
`wait`: синхронизация с дочерними

- **`int wait(int *child_status)`**
  - приостанавливает текущий процесс до завершения одного из дочерних,
  - возвращает **`pid`** завершившегося дочернего
  - if **`child_status != NULL`**, то он указывает на статус завершения дочернего

wait: синхронизация с дочерними

```
void fork9() {
   int child_status;

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



wait() : пример

- Несколько завершённых дочерних, выбираются в произвольном порядке
- Макросы WIFEXITED и WEXITSTATUS выдают информацию о статусе завершения

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Дочерний */
    for (i = 0; i < N; i++) {
        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);
    }
}
```

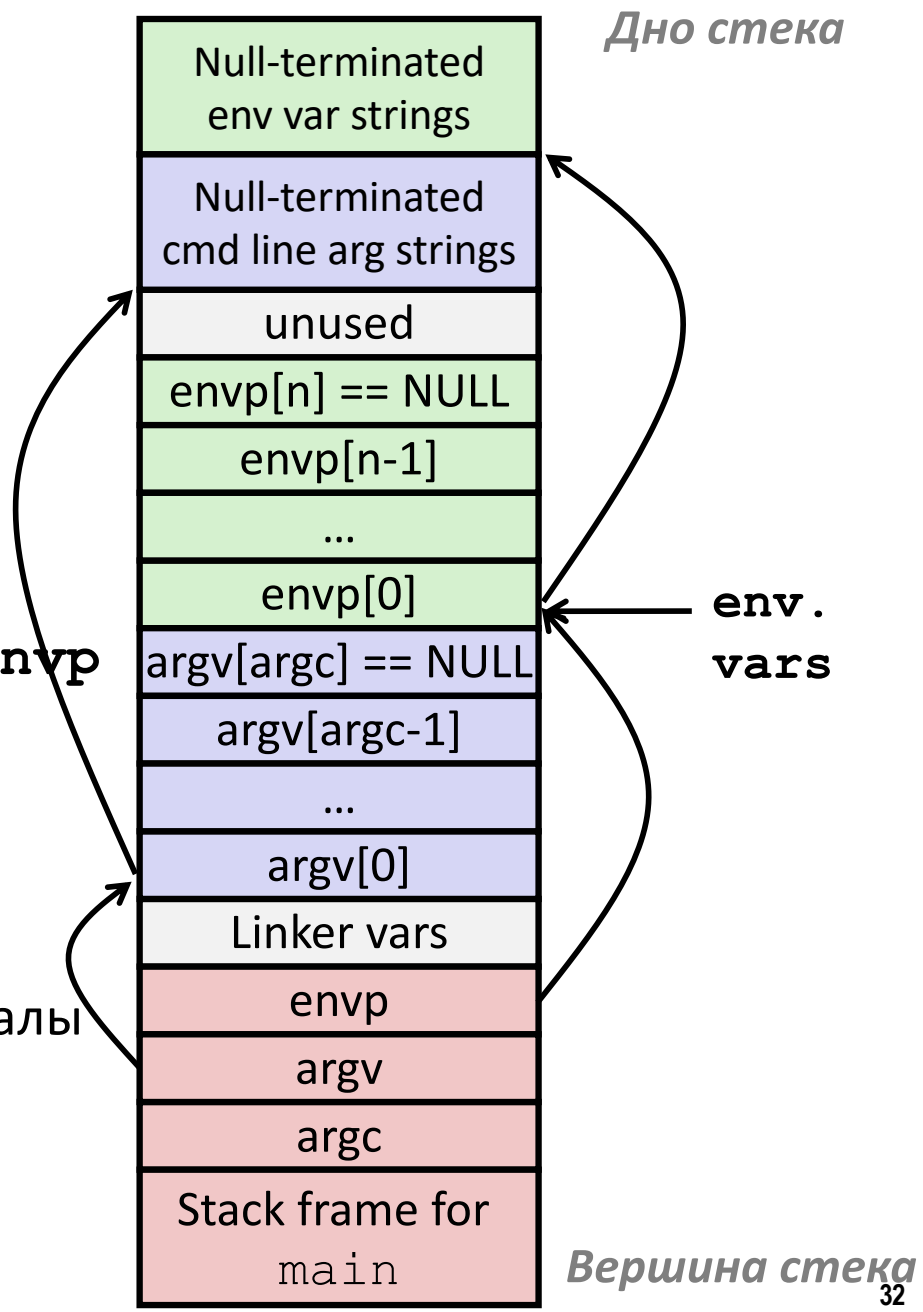
waitpid(): ожидание указанного процесса

- **waitpid(pid, &status, options)**
  - приостанавливает текущий процесс до окончания указанного

```
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); /* Дочерний */
    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 terminated abnormally\n", wpid);
    }
}
```

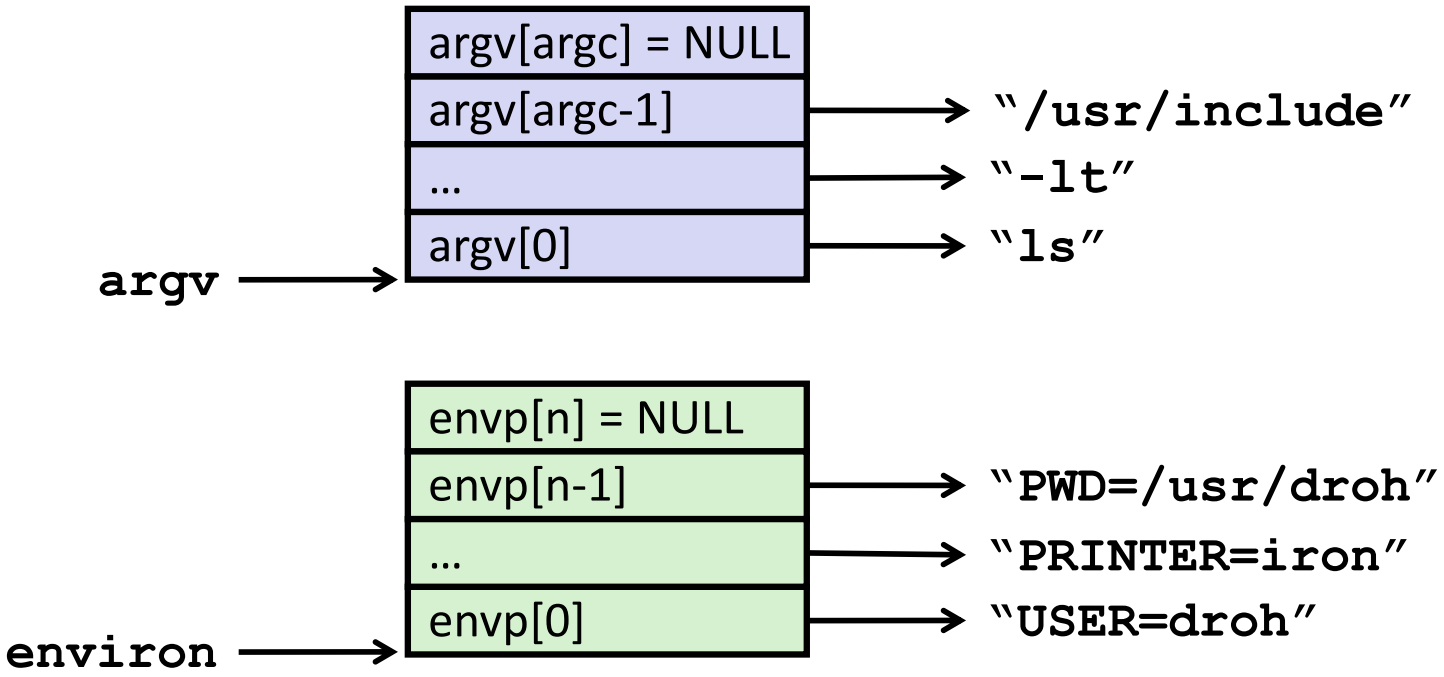
execve: загрузка исполнение программ

- `int execve(`
  `  char *filename,`
  `  char *argv[],`
  `  char *envp[]`
  `)`
- **Загружает и исполняет:**
  - Исполняемый объект из **`filename`**
  - Список аргументов из **`argv`**
  - Список переменных окружения из **`envp`**
- **Не возвращает управление**
  - если нет ошибки
- **Переписывает code, data и stack**
  - сохраняет pid, открытые файлы, сигналы
- **Переменные окружения:**
  - Строки “name=value”
  - `getenv` и `putenv`



execve: пример

```
if ((pid = Fork()) == 0) { /* дочерний исполняет задание */
    if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}
```



Сводка

- **Исключения**
  - События требующие необычного потока управления
  - Порождаются и вовне (прерывания), и внутри (ловушки и сбои)

- **Процессы**
  - В любой момент времени, в системе активны несколько процессов
  - При этом только один может исполняться на единственном ядре
  - Каждому процессу кажется, что он полностью и монопольно контролирует ЦП и память.

Сводка (продолжение)

- **Запуск процессов**
  - Вызов `fork`
  - Однажды вызванная управление возвращает дважды
- **Окончание процессов**
  - Вызов `exit`
  - Однажды вызванная управление не возвращает
- **Срезание и ожидание процессов**
  - Вызов `wait` или `waitpid`
- **Загрузка и исполнение программ**
  - Вызов `execve` (или другие варианты)
  - Однажды вызванная управление (обычно) не возвращает