Московский Физико-Технический Институт
Физтех-школа Аэрокосмических технологий
Институт Аэромеханики и Летательной Техники

МФТИ

ФАКТ

Архитектура Компьютера и Операционные Системы. Часть 2. Основы операционных систем

Лекция 3: Процессы в Unix

Новиков Андрей Валерьевич
д.ф.-м.н.

Жуковский

Процесс

- Программа + входные данные + момент исполнения

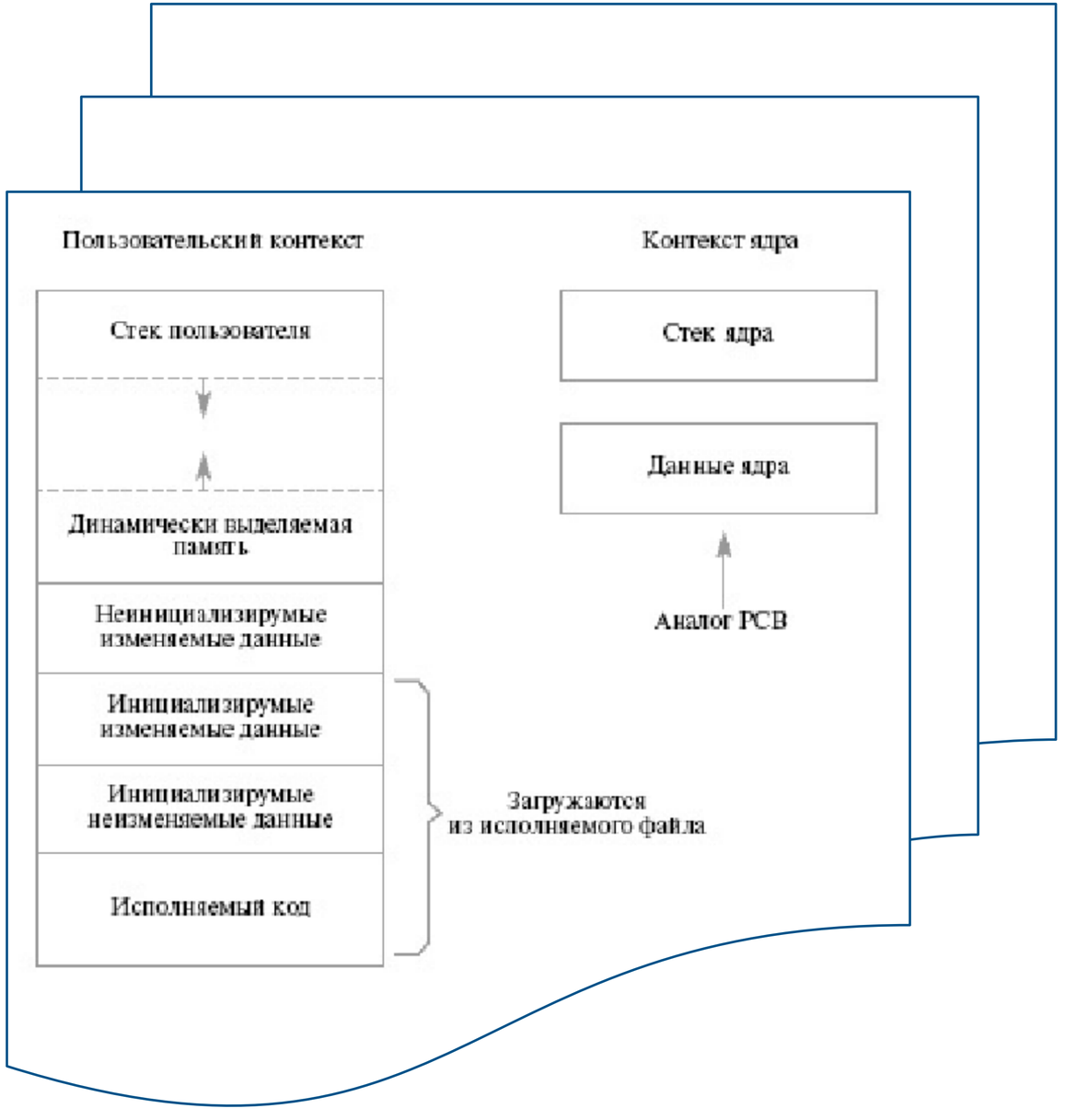
- Процесс — совокупность
  - набора исполняющихся команд,
  - ассоциированных ресурсов (выделенная память, адресное пространство, стеки, файлы, устройства ввода-вывода и т.д.),
  - текущего момента выполнения (значения регистров, программного счётчика, состояния стека, значения переменных и т.д.)

  под управлением операционной системы

- Всё что выполняется в ОС – процесс!

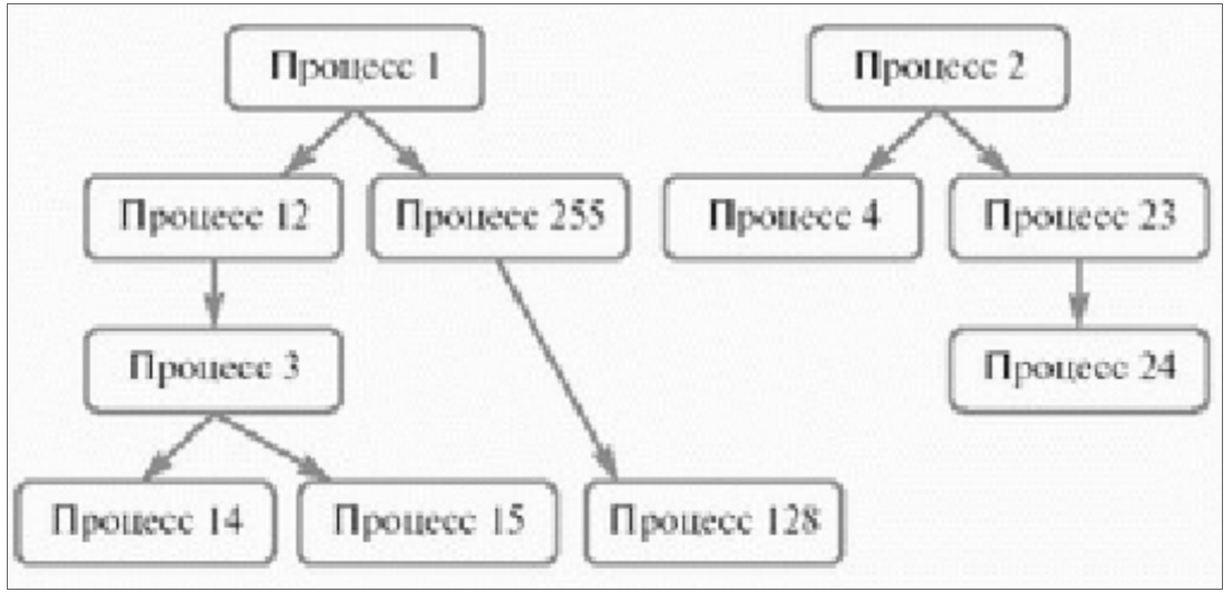
Контекст процесса

- Контекст = пользовательский + ядерный
- Контекст ядра = регистровый контекст + системный; стек ядра + данные ядра
- Данные ядра (PCB Process Control Block) :
  - идентификатор процесса PID
  - идентификатор родительского процесса PPID
  - идентификатор пользователя UID
  - идентификатор группы GID
  - ...



Иерархия процессов

- Каждый процесс порождается другим
- Создающий процесс – родитель (parent process)
- Создаваемый процесс – ребёнок, «дочерний процесс» (child process)
- Идентификатор процесса в Unix – **PID** – уникальное целое число 2^31 – 1



Команды управления процессами в Linux

- **ps**
  мгновенный список процессов, запущенных текущим пользователем в текущем терминале
- **ps –eF** (или **ps aux**)
  мгновенный список всех процессов в системе в расширенном формате

- **top**
  динамический список работающих процессов
- **htop**
  динамический список процессов с интерфейсом

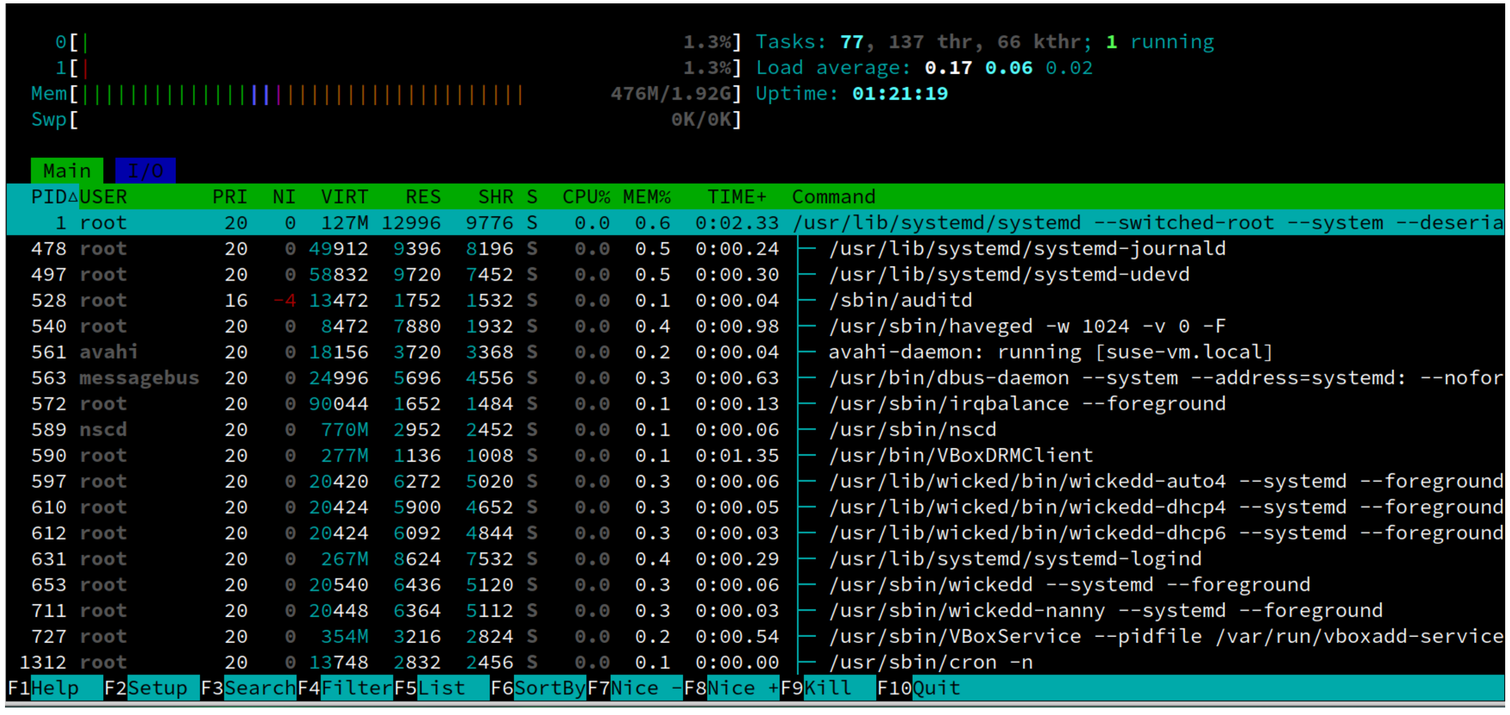


Диаграмма состояний процесса

Планирование

Системные вызовы для PID

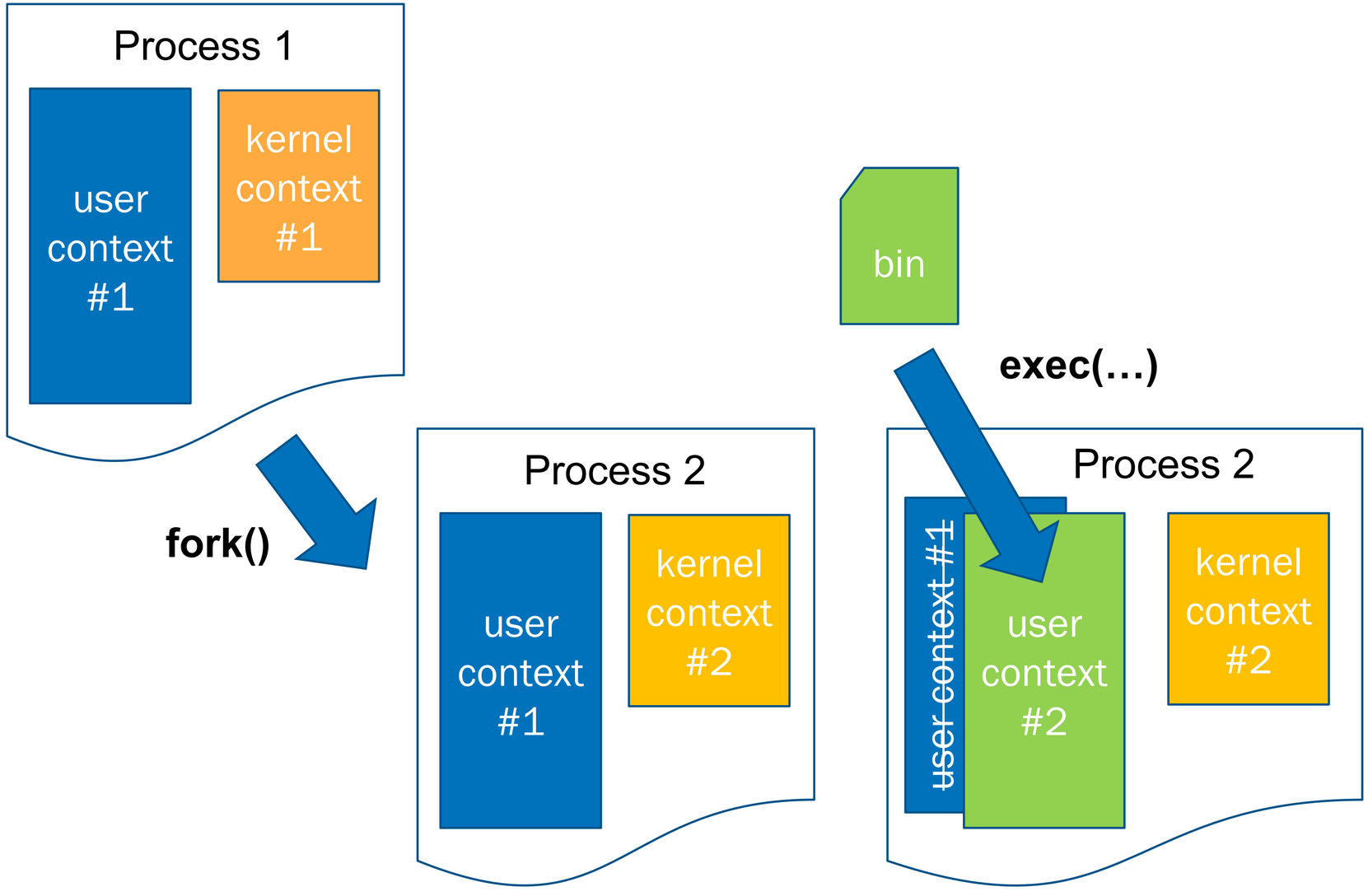
```
#include <sys/types.h>
#include <unistd.h>


pid_t getpid(void);
pid_t getppid(void);
```

- Возвращаемое значение
  - >0 – идентификатор процесса (родительского процесса)
  - -1 – ошибка

Создание процесса в Unix

1. Скопировать весь контекст (по методу **copy-on-write**)
2. Возможно настроить окружение
3. Заменить пользовательский контекст



Системный вызов создания процесса

```
#include <sys/types.h>
#include <unistd.h>


pid_t fork(void);
```

- Создаёт копию текущего процесса
- Возвращается **дважды**!
  - в родительском процессе
    > 0 – PID ребёнка
    -1 - ошибка
  - в процессе-ребёнке
    = 0 – успешное порождение

Использование fork()

```
pid_t pid = fork();
if( pid > 0 ){
   // родитель
} else if( pid == 0 ){
   // ребёнок
} else {
   // ошибка
}
```

- После выхода из fork() продолжается исполнение следующего за вызовом кода (одинакового!), как в родителе, так и в ребёнке.
- Ветви исполнения кода в родителе и ребёнке можно разделить по возвращаемому значению

Завершение процесса

```
#include <stdlib.h>


void exit(int status);
```

- Нормальное завершение процесса – закрытие файлов, опустошение буферов, перевод в состояние «закончил исполнение»
- Возврата в программу НЕТ.
- status – код завершения процесса передаётся ОС
  - 0 – 255 допустимые значения (8 бит)
  - 0 – индикация успешного завершения

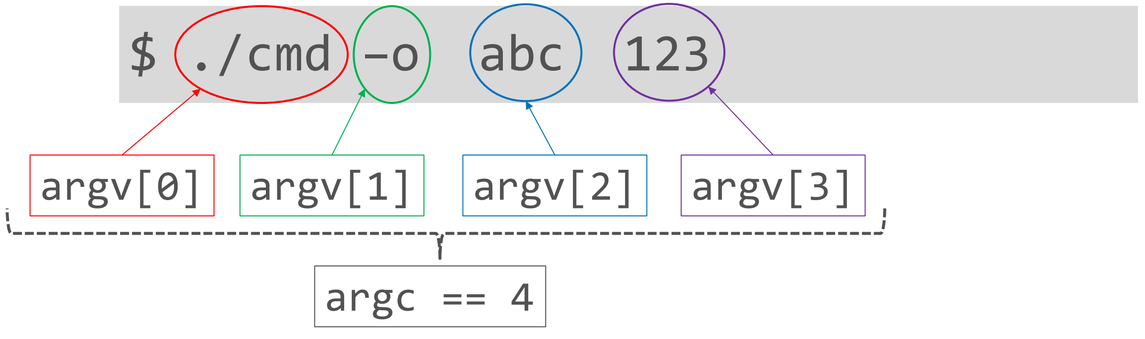
Завершение процесса

- Если родитель завершается раньше ребёнка
  → ребёнок получает PPID=1, т.е. родителем становится PID=1 (головной процесс `init` или `systemd`)
- Если родитель не запрашивает информацию о завершении
  → ребёнок остаётся в состоянии «завершил исполнение»
  = процесс-зомби (zombie, defunct)

Процесс в представлении С

```
int main(
   int argc,
   char* argv[],
   char* envp[]
){
   …
   return err_code;
}
```

- **`main()`** – точка входа
- **`return err;`** из main() ⟶ вызов exit(err)
- Параметры передаются из ОС при запуске
  - argc – количество слов-аргументов в командной строке (arguments count)
  - argv[] – массив слов в «командной строке» (arguments values)
  - envp[] – список переменных окружения, строк в виде “var=value”



Изменение контекста процесса

```
#include <unistd.h>

int execl(const char *path, const char *arg,
..., NULL);

int execlp(const char *file, const char *arg,
..., NULL);

int execle(const char *path, const char *arg,
..., NULL, char * const envp[]);


int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const
argv[]);

int execve(const char *path, char *const argv[],
char* const envp[]);
```

- **Уничтожает(!)** текущий пользовательский контекст и **заменяет** его на загружаемый из исполняемого файла
- Возвращает
  - -1 – при ошибке
  - **НЕ возвращается** при успехе
- file – имя исполняемого бинарного(!) файла, автопоиск в стандартных каталогах, перечисленных в PATH
- path – путь к исполняемому бинарному(!) файлу (каталог + имя)

Изменение контекста процесса

```
#include <unistd.h>

int execl(const char *path, const char *arg,
..., NULL);

int execlp(const char *file, const char *arg,
..., NULL);

int execle(const char *path, const char *arg,
..., NULL, char * const envp[]);


int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const
argv[]);

int execve(const char *path, char *const argv[],
char* const envp[]);
```

- **argv[]** – массив аргументов «командной строки» для исполняемой программы, который будет доступен через параметр функции main(...argv[]...)
- **arg, …, NULL** – перечисление аргументов командной строки, из которых автоматически формируется argv[]
- В обоих случаях, аргументы командной строки должны **явно включать 0-ой**, который обычно соответствует названию запускаемой программы.

Разновидности exec()

Использование exec()

```
#include <unistd.h>
int main(int argc, char* argv[], char* envp[]){
    pid_t p = fork();
    if( p == -1 ){  perror(“fork”);  return -1; }
    if( p == 0 ){
        int ret = execle(“/bin/ls”, “ls”, “-l”, NULL, envp);
        // if( ret == -1 ) {
        perror(“exec”);  retrun -2;
    }
    return 0;
}
```