

# Операционные системы

19 марта 2019 г.

# Содержание

<b>1</b>	<b>Введение</b>	<b>2</b>
1.1	Преподаватель . . . . .	2
1.2	Операционные системы . . . . .	2
1.3	Ядро и прочее . . . . .	2
<b>2</b>	<b>Процессы</b>	<b>3</b>
2.1	Немного повторения . . . . .	3
2.2	PID . . . . .	3
2.3	Calling convention . . . . .	4
2.4	Диаграмма времени жизни процесса и взаимодействия с ОС . . . . .	5
2.5	Homework . . . . .	5
2.6	Переключение контекста . . . . .	6
<b>3</b>	<b>Файлы</b>	<b>7</b>

# Лекция 1

## Введение

### 1.1 Преподаватель

Банщиков Дмитрий Игоревич  
**email:** me@ubique.spb.ru

### 1.2 Операционные системы

- Операционная система — это уровень абстракции между пользователем и машиной. Цель курса в том, чтобы объяснить что происходит в системе от нажатия кнопки в браузере до получения результата.
- Курс будет посвящен Linux, потому что иначе говорить особо не о чем. Linux - это операционная система общего назначения, для машин от самых маленьких почти без ресурсов до мощнейших серверов. Простой ответ почему Linux настолько популярен, а не Windows — в некоторых случаях он бесплатный.
- Почему полезно разрушить абстракцию черного ящика? Чтобы писать более оптимизированный и функциональный код. Иногда встречаются проблемы которые не могут быть решены без знания внутренней работы ОС.

### 1.3 Ядро и прочее

- Ядро Linux (*kernel*) — монолитное, это оправдано для ядра, но уязвимость одной части ядра ставит в угрозу все остальные части.
- Микроядерные ОС - альтернатива монолитным (мы не будем их изучать), но с ними сложно работать, потому что протоколы общения между частями требуют ресурсов.
- *UNIX-like* системы - это системы предоставляющие похожий на *UNIX* интерфейс.

# Лекция 2

## Процессы

TODO from day02

### 2.1 Немного повторения

- *fork()* - для того чтобы создать новый процесс
- *wait(pid)* - ждем процесс
- *exit()* - завершаемся
- *SIGKILL* - принудительное завершение другого процесса ( **\$ kill** )

### 2.2 PID

- У каждого *PID* есть *parentPID* (*PPID*)
- **\$ ps** - позволяет посмотреть специфичные атрибуты процесса
- Процесс *init(pid 0)* создается ядром и выступает родителем для большинства процессов, созданных в системе
- Можно построить дерево процессов ( **\$ pstree** )

Процесс делает *fork()*. Возможны 2 случая:

1. Процесс не делает *wait(childpid)*

Зомби-процесс (*zombie*) - когда дочерний процесс завершается быстрее, чем вы сделаете *wait*

2. Процесс завершается, что происходит с дочерним процессом?

Сирота (*orphan*) - процесс, у которого умер родитель. Ему назначается родителем процесс с *pid 1*, который время от времени делает *wait()* и освобождается от детей

*PID* - переиспользуемая вещь (таблица процессов)

## 2.3 Calling convention

\$ **man syscall** - как вызываются *syscall*

syscall.h

```
#ifndef SYSCALL_H
#define SYSCALL_H

void IFMO_syscall();

#endif
```

syscall.s

```
.data

.text
.global IFMO_syscall

IFMO_syscall:
    movq $1, %rax
    movq $1, %rdi
    movq $0, %rsi
    movq $555, %rdx
    syscall
    ret
```

main.c

```
#include "syscall.h"

int main() {
    IFMO_syscall();
}
```

Что здесь происходит?

1. Вызываем `write()`
2. Просим ядро записать 555 байт начинающихся по адресу 0 в файловый дескриптор №1 (*stdout* - №1, *stdin* - №2, *stderr* - №3)
3. Ничего не происходит, так как:  
*write(1, NULL, 555)* возвращает -1 (*EFAULT* - Bad address)

Как со всем этим работать?

- \$ **strace** - трассировка процесса (подсматриваем за процессом, последовательность *syscall* с аргументами и кодами возврата)

Если *syscall* ничего не возвращает, то в выводе пишется ? вместо возвращаемого значения

- **\$ man errno** - ошибки

Если делаем *fork()* - проверяем код возврата (хорошая практика)

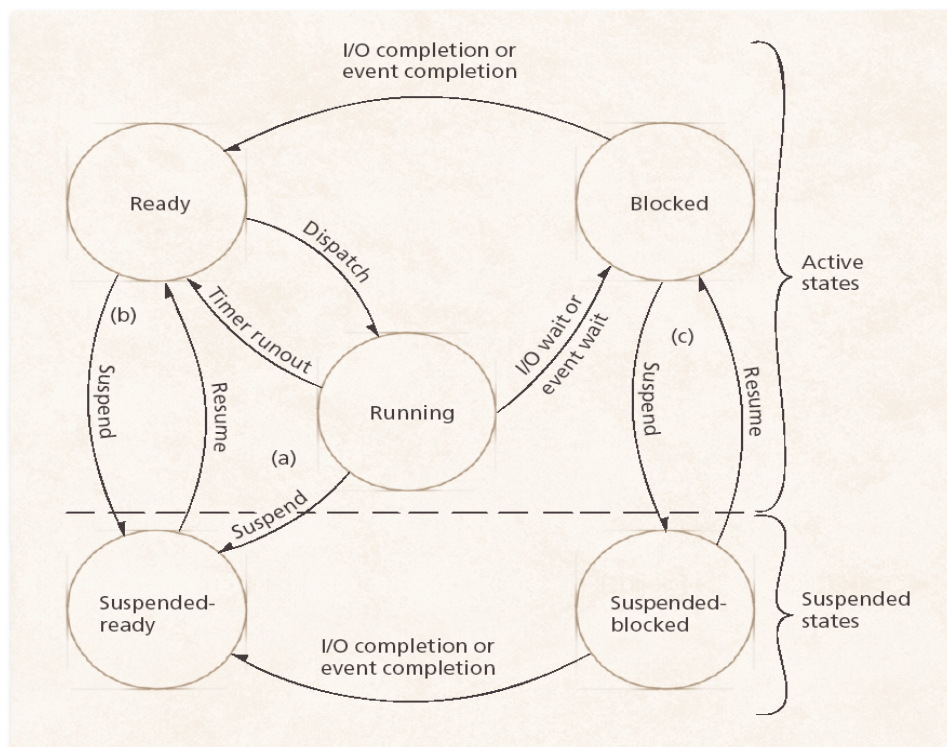
*char\* strerror(int errnum)* - возвращает строковое описание кода ошибки

Почему *char\**, а не *const char\**? Потому что всем было лень.

*thread\_local* - решение проблемы: переменная с ошибкой - общая для каждого потока

- До *main()* и прочего (конструкторы) происходит куча всего (*munmap*, *mprotect*, *mmap*, *access*) - размещение процесса в памяти и т.д.
- Программа не всегда завершается по языковым гарантиям (деструкторы)
- **\$ ptrace** - позволяет одному процессу следить за другим (используется, например, в *GDB*)

## 2.4 Диаграмма времени жизни процесса и взаимодействия с ОС



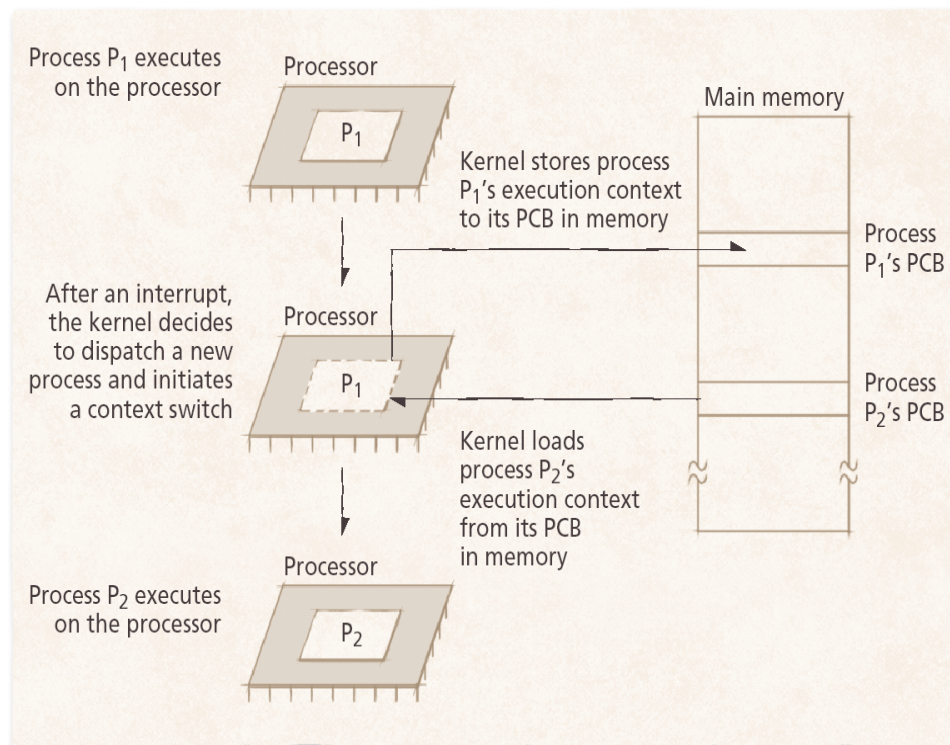
## 2.5 Homework

Написать shell-интерпретатор

- Читать из stdin
- В дочернем процессе execve()
- В родительском процессе wait()
- Сдавать через гитхаб

## 2.6 Переключение контекста

Здесь иллюстрируется иллюзия многозадачности



# Лекция 3

## Файлы

TODO