Операционные системы

19 марта 2019 г.

Содержание

1 Введение **2**
1.1 Преподаватель 2
1.2 Операционные системы 2
1.3 Ядро и прочее 3

2 Процессы **4**
2.1 Немного повторения 4
2.2 PID 4
2.3 Calling convention 5
2.4 Диаграмма времени жизни процесса и взаимодействия с ОС 7
2.5 Homework 7
2.6 Переключение контекста 7

3 Файлы **9**

Лекция 1

Введение

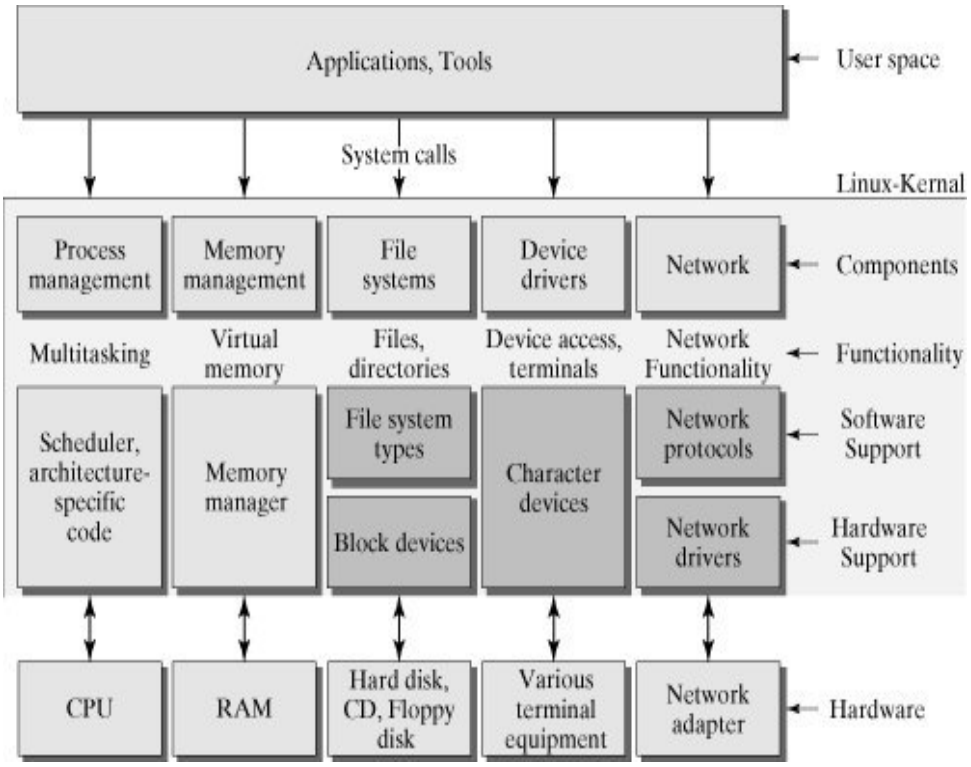
1.1 Преподаватель

Банщиков Дмитрий Игоревич
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 Линуксу, потому что иначе говорить особо не о чем. Линукс это операционная система общего назначения, для машин от самых маленьких почти без ресурсов до мощнийших серверов. Простой ответ почему линукс настолько популярен нежели виндоус в некоторых случаях - он бесплатный.
Почеум полезно разрушить абстракцию черного ящика? Чтобы писать более оптимизированный и функци ональный код. Иногда встр ечаются проблемы которые не могут быть решены без знания внутренней работы ОС.

1.3 Ядро и прочее



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

Процесс - эксзкмпляр запущенной программы. Процессы должны уметь договариваться чтобы сосуществовать, но в то же времся не знать друг о друге и владеть монополией на ресурс машины. С точки зрения ОС процесс - это абстракция, позволяющая абстрагироваться от внутренностей процеса. С точки зрения программиста процесс = абстракция, которая позволяет думать что мы монопольно владеем ресурсами машины.
На момент выполнения процесс можно охарактеризовать полным состоянием его памяти и регистров. Чтобы приостановить процесс нам нужно просто сохранить его 'отпечаток', а чтобы возобновить нужно загрузить его память и регистры. одноядерках.
Батч-процессы типа сборки, компиляции не требуют отзывчивости пока жрут ресурсы.

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 назначется 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_syscal();
}
```

Что здесь просходит?

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)

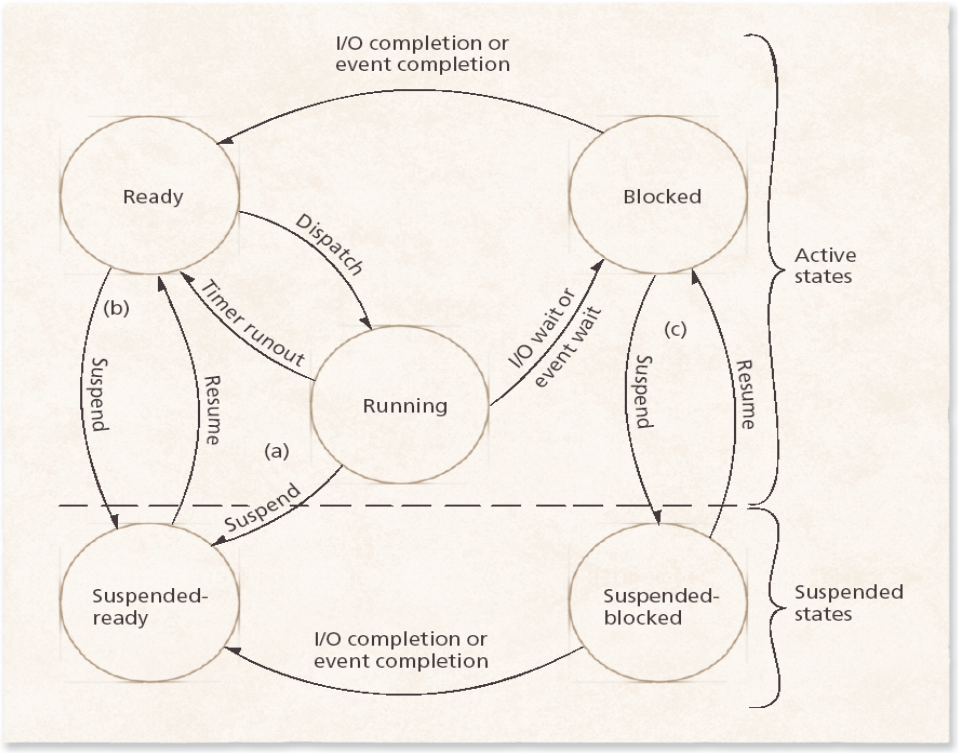
  char strerror(int errnum)* - возвращает строковое описание кода ошибки

  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



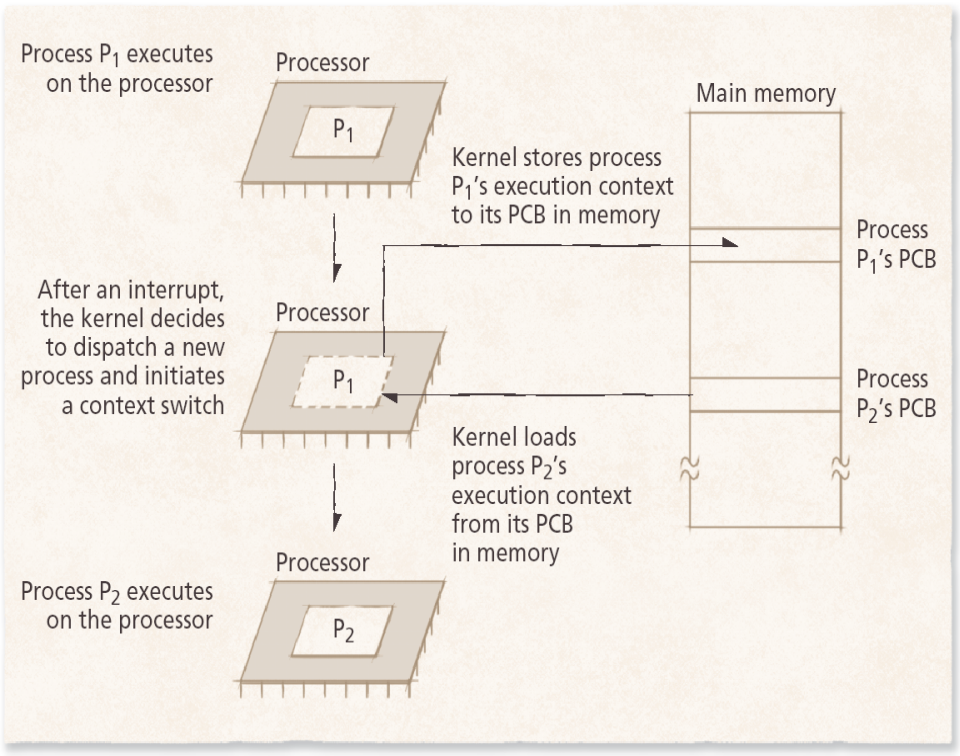
2.5 Homework

Написать shell-интерпретатор

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

2.6 Переключение контекста

Шедулер ОС раскидывает процессы и создает иллюзию одновременного выполнения на Здесь иллюстрируется иллюзия многозадачности


Process P_1 executes on the processor
Processor
P_1
Main memory
Kernel stores process P_1's execution context to its PCB in memory
Process P_1's PCB
After an interrupt, the kernel decides to dispatch a new process and initiates a context switch
Processor
P_1
Process P_2's PCB
Kernel loads process P_2's execution context from its PCB in memory
Processor
Process P_2 executes on the processor
P_2

Лекция 3

Файлы

TODO