

Основы системного программирования

Ввод-вывод и работа с файлами.
Многозадачность и процессы. Трассировка
программ

Гирик Алексей Валерьевич

Университет ИТМО
2021

Материалы курса

- Презентации, материалы к лекциям, литература, задания на лабораторные работы
 - shorturl.at/gjABL



Ввод-вывод и работа с файлами

Подсистема ввода-вывода

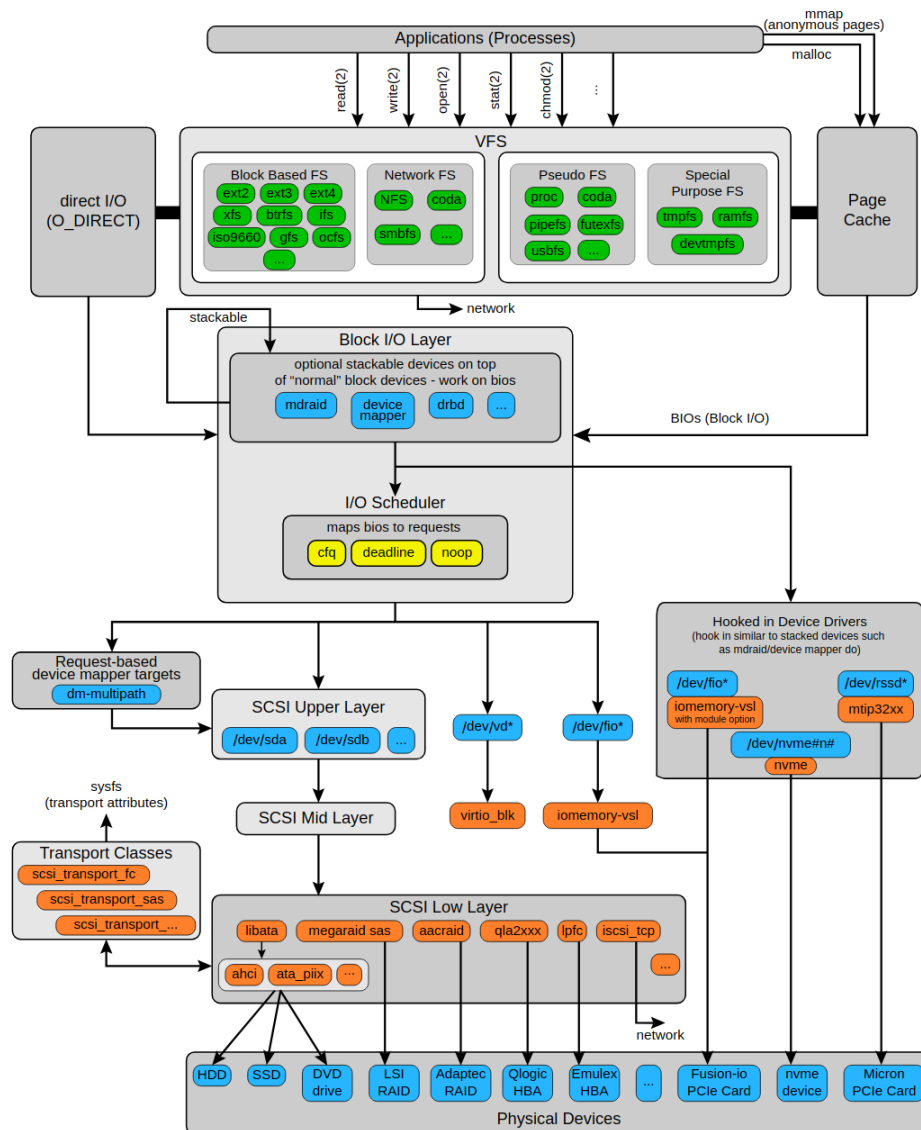
- любой POSIX-совместимой системе присущи отличительные особенности:

- модель ввода-вывода на основе файлов

- представление устройств и системных объектов в виде файлов
- ограниченный набор универсальных файловых операций

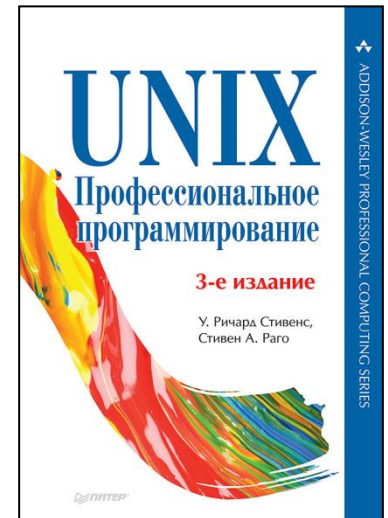
- единое дерево каталогов с точками монтирования, куда подключаются различные файловые и псевдофайловые системы

- в Linux - VFS



Основные операции с файлами и каталогами

- файлы
 - создание, удаление
 - чтение, запись данных
 - текст
 - бинарные данные
 - чтение, изменение атрибутов
- каталоги
 - создание, удаление
 - чтение содержимого
 - чтение, изменение атрибутов



Главы 3 – 5

Ввод-вывод с помощью системных вызовов

- **ОСНОВНЫЕ СИСТЕМНЫЕ ВЫЗОВЫ:** `open()`, `read()`, `write()`, `lseek()`, `close()`
- **открытые файлы идентифицируются с помощью дескрипторов (целое число типа `int`)**

```
int fd = open("hello.txt", O_RDONLY);  O_RDONLY: file mở chỉ dành để đọc
if (fd < 0) {
    // Ошибка
}
else {
    // ОК
    ...
    close(fd);      // дескрипторы необходимо закрывать!
}
```

Что нужно знать про read() и write()

- ОСНОВНЫЕ ВЫЗОВЫ ДЛЯ ЧТЕНИЯ/ЗАПИСИ ДАННЫХ ИЗ/В файлы, сокеты, каналы, ...
- возвращают количество прочитанных/записанных байтов или -1

```
#include <unistd.h>
```

```
ssize_t read(int fd, void *buf, size_t count);  
ssize_t write(int fd, void *buf, size_t count);
```

hàm read() đọc count bytes từ tệp được liên kết với bộ mô tả tệp đang mở (descriptor), fd vào bộ đệm được trỏ tới bởi buf

Ввод-вывод с помощью стандартной библиотеки C

- `fopen()`, `fgets()`, `fputs()`, `fread()`, `fwrite()`, `fseek()`, `fclose()`, ...
- открытые файлы идентифицируются с помощью указателей на структуру `FILE`

мở tên file được trỏ tới theo tên tệp (filename) bằng chế độ đã cho (mode)

```
FILE *fp = fopen("hello.txt", "r");  
if (!fp) {  
    // Ошибка  
}  
else {  
    // OK  
    ...  
    fclose(fp);  
}
```

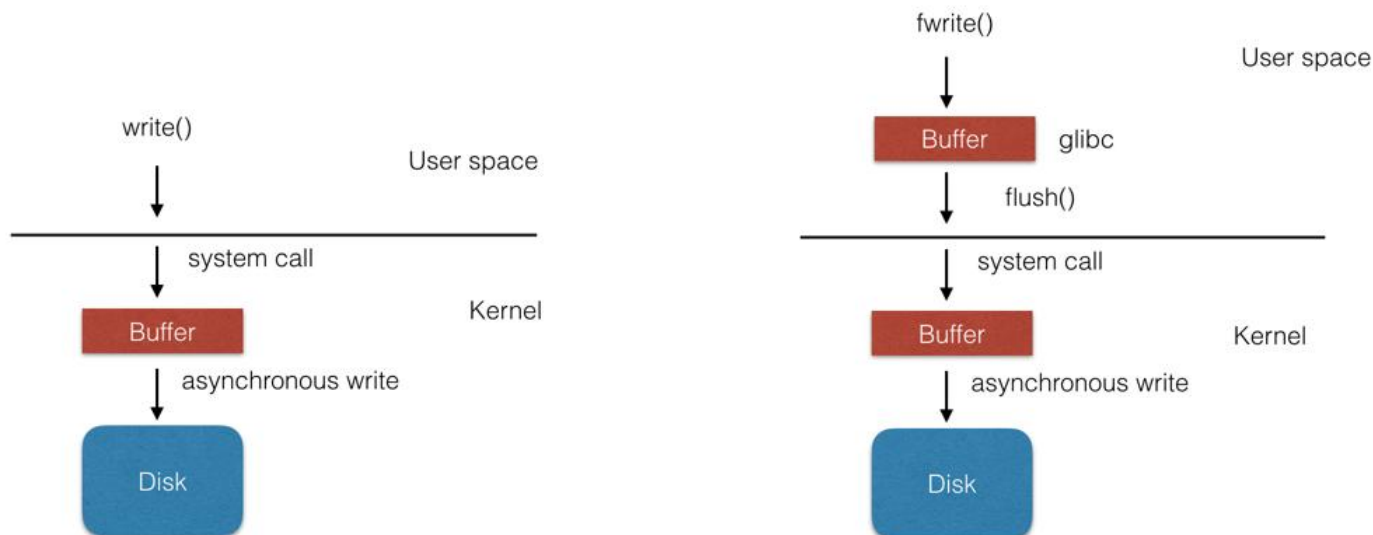
"r" : mở tệp để đọc (tệp phải tồn tại)
"w" : tạo 1 file trống để viết (nếu trùng tên thì dữ liệu cũ bị xóa và xem như tệp trống mới)
"a" : thêm vào file, ghi thêm dữ liệu vào cuối file, file sẽ được tạo nếu nó k tồn tại
"r+" : mở file để đọc và ghi (file phải tồn tại)
"w+" : tạo file trống để đọc và viết
"a+" : mở file để đọc và viết

FILE *fopen(const char *filename, const char *mode)

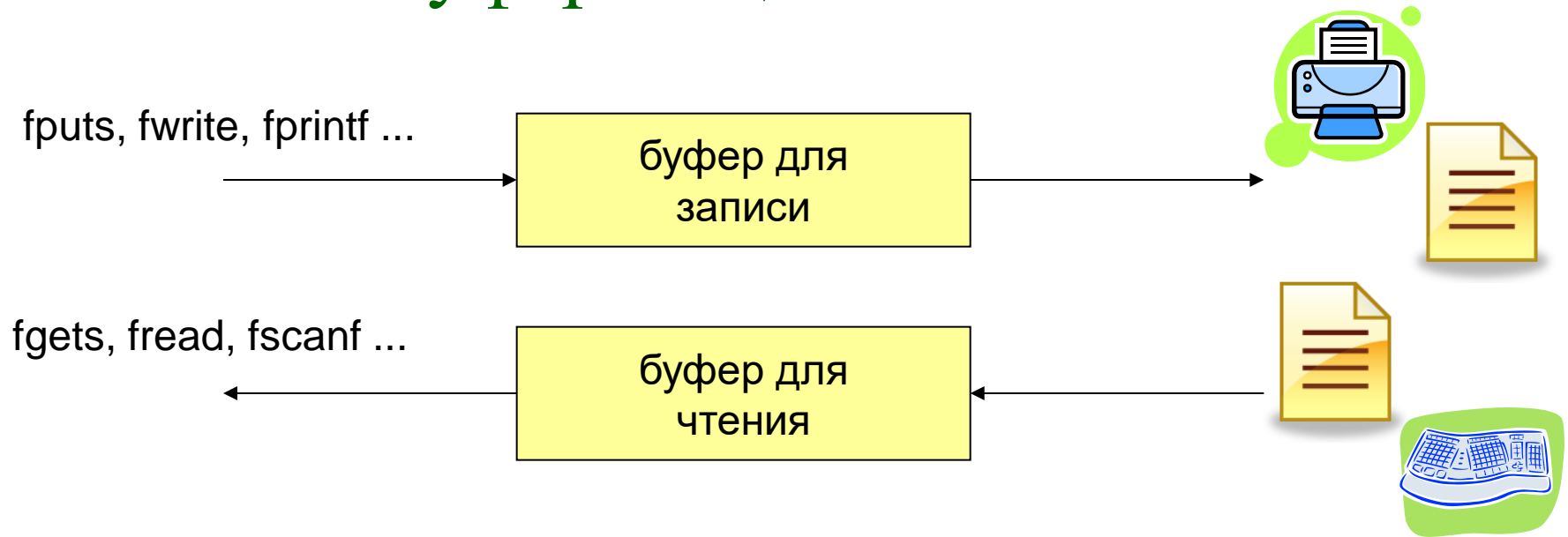
// Указатели на файлы необходимо закрывать!

Буферизация при вводе-выводе

- буферизация в ядре – page cache, гранулярность – блок ФС (фактически – страница)
- буферизация в библиотеке C, гранулярность – настраиваемая с помощью `setvbuf()`



Буферизация потоков



увеличение скорости операций чтения/записи за счет сокращения количества обращений к устройству (диску, консоли и т.д.)

Управление буферизацией

```
int setvbuf(FILE *stream, char *buffer,  
            int mode, size_t size)
```

...

```
char *buffer = (char*)malloc(1024*1024);  
setvbuf(f, buffer, _IOFBF, 1024*1024);
```

...

```
setvbuf(f, NULL, _IONBF, 0);
```

Пара слов о типах файлов

текстовые (plain text)

бинарные (binary)

по формату (типу данных, которые хранятся в файле):

- исполняемые (программы)

- картинки

- видео

- аудио

- документы

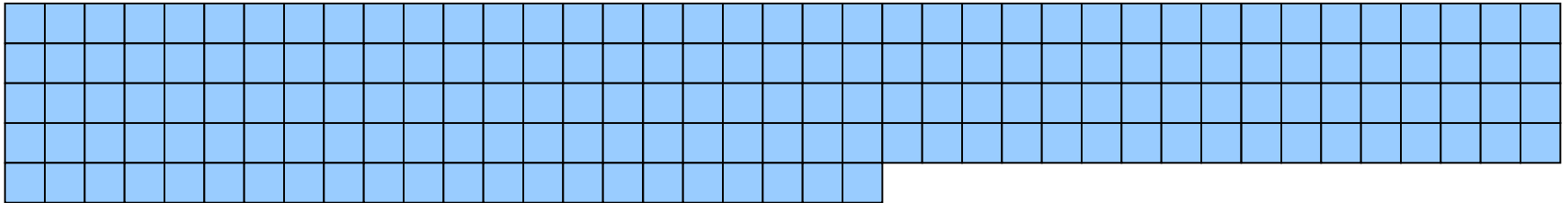
- псевдофайлы (специальные файлы)

- ...

Бинарные файлы

binary – файлы, содержащие любые данные (и текстовые, и не текстовые)

■ – байт (число без знака в диапазоне 0 .. 255)



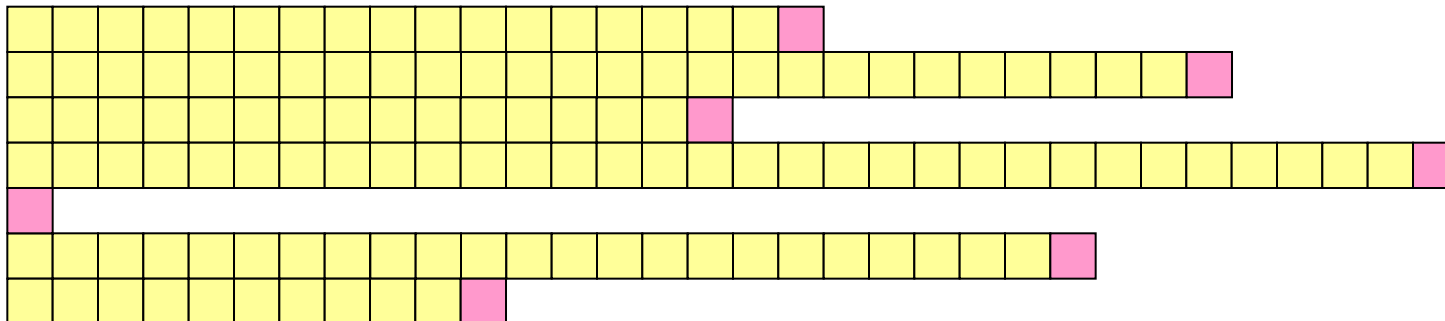
http://ru.wikipedia.org/wiki/Двоичный_файл

Текстовые файлы

plain text – файлы в восьмибитной кодировке, содержащие только текстовые данные

■ – текстовый символ (буква, цифра, знак препинания, пробел, табуляция ...)

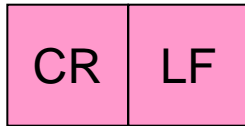
■ – управляющий символ - признак перевода строки



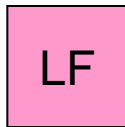
http://ru.wikipedia.org/wiki/Plain_text

Признак перехода на новую строку

Windows



UNIX, Linux



CR = Carriage Return (возврат каретки)

ASCII code, 13 (0xD) `'\r'`

LF = Line Feed (перевод строки)

ASCII code 10 (0xA), `'\n'`

Традиционная ошибка

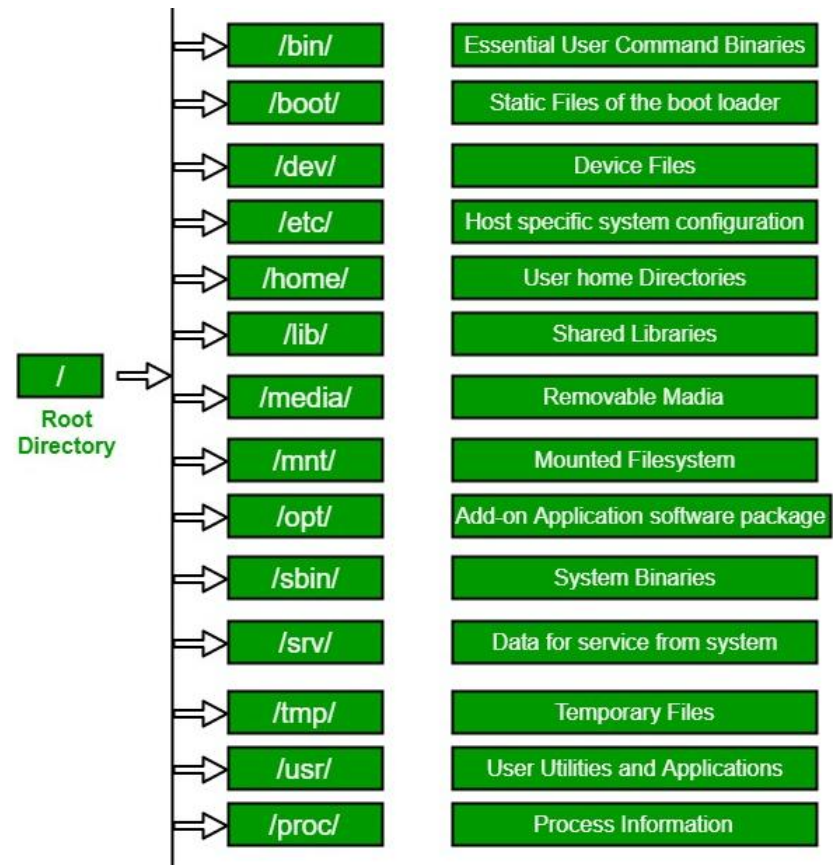
- посимвольное чтение из файла с помощью `fgetc()`
- не настолько неэффективно, как может показаться, но в случаях, когда быстродействие важно, следует использовать `mmap()`

Построчное чтение файла

- можно использовать `fgets()`
 - если строки длинные, то можно не прочитать строку целиком – не катастрофа, но может создать неудобство
 - какого размера требуется буфер?
- проще пользоваться `getline()`
 - тем более, что теперь она входит в POSIX.1

Текущий (рабочий) каталог

- файл программы размещается в некотором каталоге
 - для программ, установленных в системе есть некоторый набор более-менее стандартных каталогов
 - подробнее в FHS (<https://ru.wikipedia.org/wiki/FHS>)
- каждому процессу ОС определяет текущий рабочий каталог
 - можно получить с помощью `getcwd()`



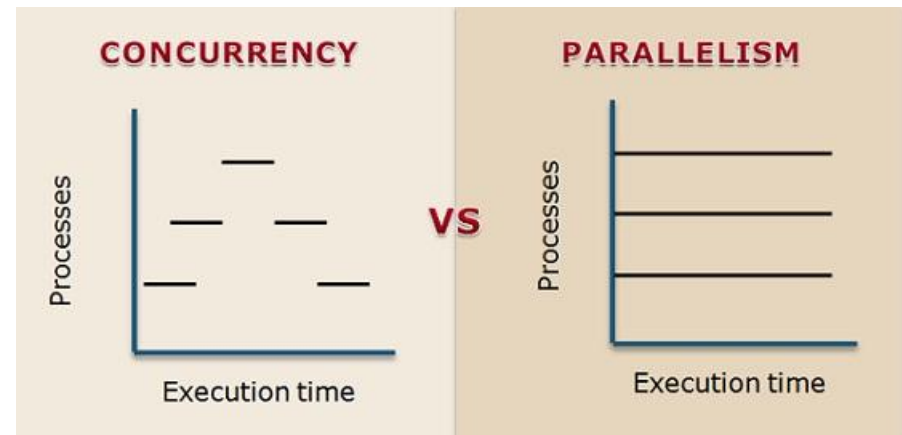
Обход дерева каталогов

- "сложный" вариант
 - рекурсивный обход с помощью `chdir()/readdir()`
- "простой" вариант
 - использование функции `ntfw()`

Многозадачность и процессы

Термины

- Параллелизм (parallelism)
- Конкурентность (concurrency)
- Многозадачность (multitasking)
- Многопроцессность (multiprocessing)
- Многопоточность (multithreading)
- Асинхронная обработка (asynchronous processing)



Многозадачность

Linux – многопользовательская многозадачная ОС
общего назначения

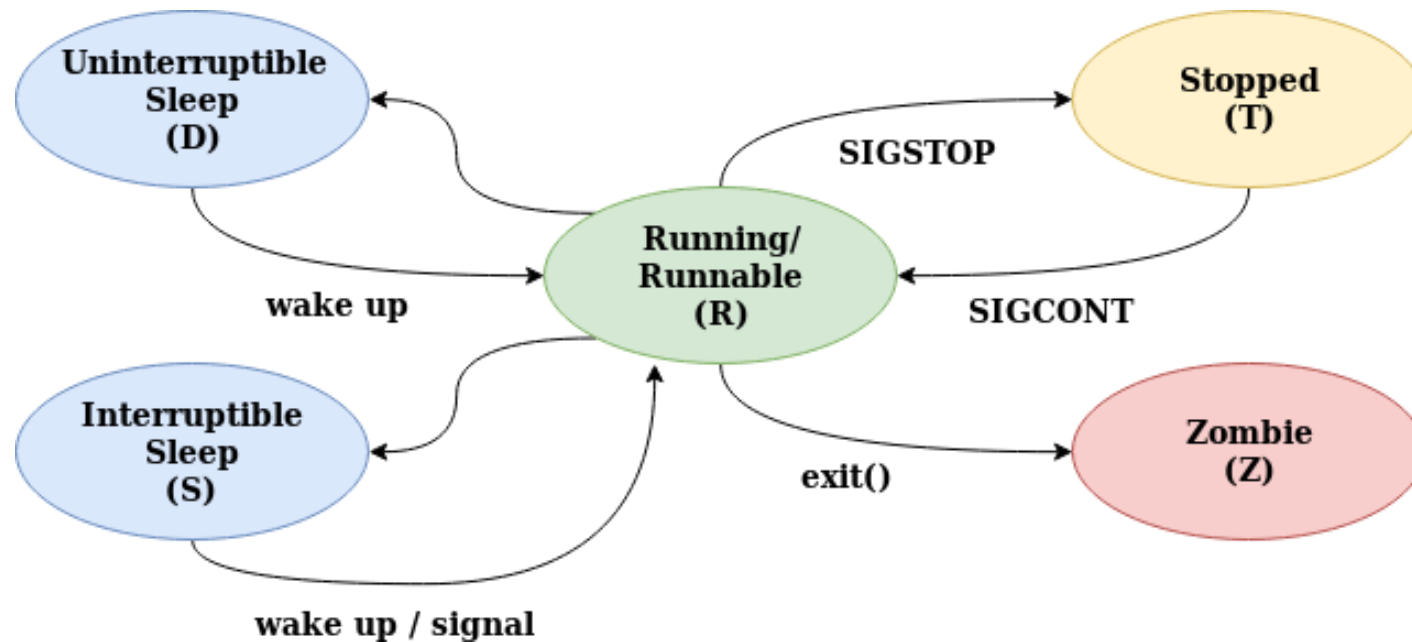
- задача (task)
- **программа (program)**
- **процесс (process)**
- поток (thread)
- задание (job)
- сессия (session)
- группы
 - процессов (process groups)
 - потоков (thread groups)

Процессы

Процесс – загруженная в память программа, единица учета системных ресурсов в ОС

- состояния процесса
- разделение памяти и адресное пространство процесса
 - MMU и ОС отображают страницы виртуального адресного пространства на страницы оперативной (физической) памяти
- изоляция процессов
 - ограничение и учет ресурсов: chroot, namespaces, cgroups

Состояния процесса в ОС Linux



Квант времени выполнения процесса

Квант времени выполнения (*time slice, time quantum*) – время, в течение которого процесс может исполняться на ядре процессора, прежде, чем диспетчер ОС инициирует переключение контекста

- в большинстве ОС – переменная величина, определяется динамически планировщиком процессов (обычно от десятков до сотен миллисекунд)
- в Linux при использовании алгоритма Completely Fair Scheduler квант по умолчанию составляет от 0.75 мс до 6 мс (от `sched_min_granularity_ns` до `sched_latency_ns`)

Контекст процесса и переключение контекста

Переключение контекста (*context switch*) – совокупность действий, которые необходимо выполнить ядру ОС для того, чтобы перестать выполнять код одного процесса и начать выполнять код другого процесса

- сохранение/восстановление регистров
- сброс TLB (?) и конвейера процессорного ядра
- как следствие: увеличение промахов кэшей всех уровней

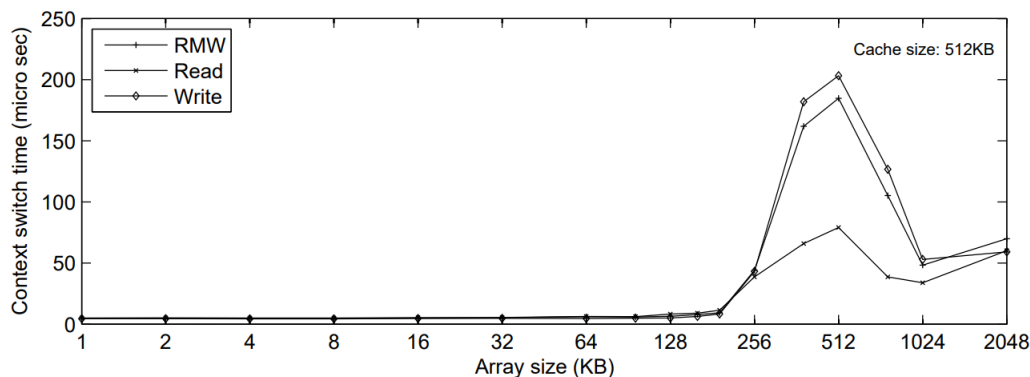
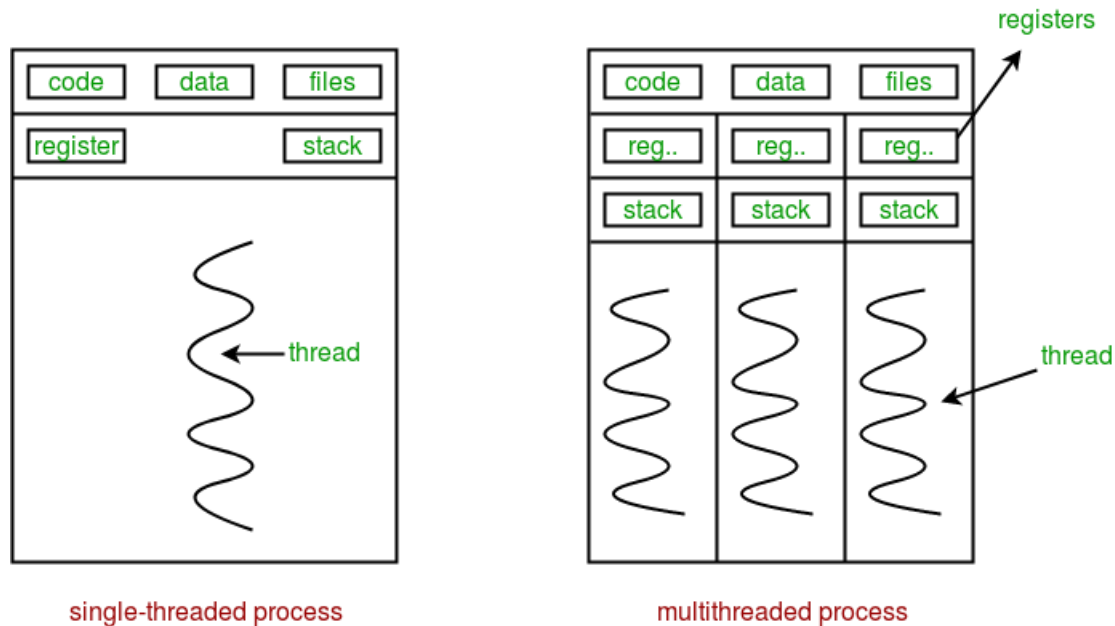


Figure 1: The effect of data size on the cost of the context switch

Процессы и потоки

- Несколько упрощая ситуацию, можно считать, что процесс – это некоторый "контейнер" для ресурсов и внутри каждого процесса выполняется как минимум один поток (thread)
 - эта упрощенная модель не соответствует действительности как минимум для Linux, как максимум – для многих POSIX-совместимых ОС



Управление процессами

Вопросы, связанные с управлением процессами:

- создание процессов
- завершение процессов
 - "принудительное" и "добровольное"
- получение информации о результате выполнения процесса
- сигналы
- взаимодействие между процессами
 - каналы
 - сокеты и другие IPC (разделяемая память, очереди сообщений, семафоры)
- организация групп процессов, сессий и другие элементы управления заданиями
 - создание процессов-демонов

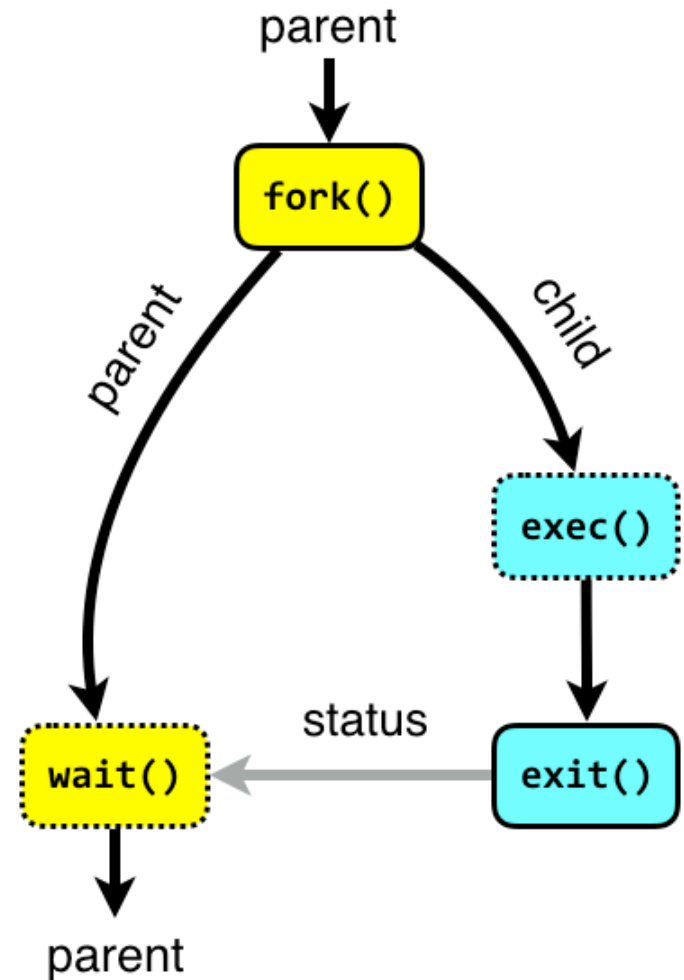
Создание нового процесса

Системный вызов fork()

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

```
#include <unistd.h>
```

```
pid_t fork(void);
```



Создание нового процесса

```
pid_t child;
if ((child = fork()) < 0) {
    // ошибка
}
else if (child == 0) {
    // внутри нового процесса
}
else {
    // внутри родительского процесса
}
```

Что наследует порожденный процесс

- окружение
- открытые дескрипторы
- umask
- текущий рабочий каталог и корневой каталог
- приоритет
- управляющий терминал
- маску сигналов, диспозицию и обработчики сигналов
- права (реальный и эффективный UID и GID)

Что у родителя и потомка разного?

- ID процесса (PID) и ID родительского процесса (PPID)
- PID потомка гарантированно отличается от ID любой существующей группы процессов
- статистика использования системных ресурсов
- если в родительском процессе ожидали доставки сигналы, в потомке они сбрасываются
- блокировки файлов, установленные в родительском процессе, не дублируются в потомке

Идентификация процесса

- ID процесса и родительского процесса можно получить с помощью системных вызовов `getpid()` и `getppid()`
- Если родительский процесс завершается раньше дочернего, то новым родителем становится процесс с `ID == 1`

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

```
#include <unistd.h>
```

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

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

Запуск новой программы внутри процесса

С помощью семейства системных вызовов `exec()`:

```
#include <unistd.h>
```

```
int execl(const char *path, char *const argv[]);  
int execlp(const char *file, char *const argv[]);  
int execlpe(const char *file, char *const argv[],  
            char *const envp[]);
```

- возврат из `exec()` возможен только в случае ошибки
- ID процесса не меняется
- в `argv[0]` может быть передана произвольная строка

Какие атрибуты наследуются процессом при вызове `exec`

- дескрипторы открытых файлов и каталогов, кроме тех, которые имели флаг `O_CLOEXEC`
- `umask`
- рабочий и корневой каталоги
- приоритет
- ID процесса, родительского процесса и группы процесса
- маска сигналов, диспозиция и обработчики сигналов, а также ожидающие доставки сигналы!
- права (действительные и эффективные `UID/GID`)
- блокировки файлов
- статистика использования ресурсов

Запуск новых программ

- Идиома "fork and exec"
 - fork не блокирует родительский процесс, используется copy-on-write
 - vfork() – запуск с блокировкой родителя

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

```
#include <unistd.h>
```

```
pid_t fork(void);
```

Завершение процесса

```
#include <stdlib.h>
```

```
void exit(int status);
```

```
void _Exit(int status);
```

```
#include <unistd.h>
```

```
void _exit(int status);
```

Регистрация обработчиков завершения

atexit() и on_exit()

```
#include <stdlib.h>
```

```
int atexit(void (*function) (void) );
```

```
int on_exit(void (*function) (int, void*),  
            void *arg);
```

Когда что использовать

- в обычных ситуациях `exit()`
 - если в порожденном процессе вызов `exec()` завершается неудачей, тогда использовать `_exit()`, чтобы избежать повторных сбросов буферов файлов, открытых в родительском процессе
- набор кодов завершения (8 bit)
 - `EXIT_SUCCESS`, `EXIT_FAILURE`
 - либо определить свой набор кодов

При запуске из скрипта оболочки

```
#!/bin/bash
```

```
some_program -x -y -z in.txt out.txt
```

```
if [[ $? != 0 ]]
```

```
then
```

```
    # ошибка!
```

```
fi
```


Получение статуса завершения процесса

- после завершения процесса ядро сохраняет статус завершения и сведения об использовании ресурсов процессом
- процесс, который завершился, но статус которого не был получен родителем, становится *зомби*



Ожидание завершения

- Системный вызов `wait()` ожидает завершения любого порожденного процесса
- возвращает `-1`, если порожденных процессов не осталось

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

```
#include <sys/wait.h>
```

```
pid_t wait(int *wstatus);
```

Ожидание завершения

- Системный вызов `waitpid()` ожидает завершения любого или заданного порожденного процесса

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

```
#include <sys/wait.h>
```

```
pid_t waitpid(pid_t pid, int *wstatus,  
              int options);
```

`pid < -1` – ждать завершения процесса с ID группы `pid`

`pid = -1` – ждать завершения любого порожденного процесса

`pid = 0` – ждать завершения любого процесса с ID группы родителя

`pid > 0` – ждать завершения процесса с заданным `pid`

- опция `WNOHANG` позволяет делать неблокирующую проверку

Получение информации о ресурсах

- Системные вызовы `wait3()` и `wait4()` позволяют получить информацию о ресурсах, которые потребил дочерний процесс:

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
```

```
pid_t wait3(int *wstatus, int options,
            struct rusage *rusage);
```

```
pid_t wait4(pid_t pid, int *wstatus, int options,
            struct rusage *rusage);
```

Дополнительные способы запустить процесс в Linux

■ system

```
#include <stdlib.h>
int system(const char *command);
```

■ popen

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

■ posix_spawn

```
#include <spawn.h>
int posix_spawn(pid_t *pid, const char *path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *attrp,
    char *const argv[], char *const envp[]);
```

Правда о запуске процессов в Linux

- Процессы и потоки в Linux запускаются с помощью системного вызова `clone()`

```
#define _GNU_SOURCE
```

```
#include <sched.h>
```

```
int clone(int (*fn) (void *), void *stack,  
         int flags, void *arg, ...);
```

Приоритет процесса

- Процесс может установить относительный приоритет с помощью системного вызова `nice`

```
#include <unistd.h>
```

```
int nice(int inc);
```

Трассировка программ

Трассировщики

- Трассировщики – инструментальные программы, позволяющие отслеживать последовательность вызовов функций в программах
- Для Linux актуальны
 - `ltrace`
 - использует `ptrace()`
 - `strace`
 - использует `ptrace()`
 - `stap` (System Tap)
 - использует eBPF bytecode

Трассировка библиотечных вызовов

- Для трассировки вызовов из динамических библиотек используется ltrace
 - не работает с PIE, при компиляции требуется ОТКЛЮЧИТЬ:

```
$ gcc -no-pie -o test test.c
```

Трассировка системных вызовов

- Для трассировки системных вызовов используется `strace`
 - можно подключиться к уже работающему процессу с помощью `-p <pid>`

Задание на дом

■ ЧИТАТЬ

- Стивенс, Раго. UNIX. Профессиональное программирование
 - главы 3 – 5, 7, 8
- Лав. Linux: Системное программирование
 - главы 2, 3, 4
 - man 2 open, close, read, write
 - man ltrace
 - man strace
 - man fork
 - man clone (бегло просмотреть)

