

# Системное программирование

Лекция 3

Процессы

# Многозадачность

**Многозадачность** – способность ОС управлять выполнением нескольких задач одновременно.

**Кооперативная многозадачность** – способ реализации многозадачности, при котором моменты передачи управления от одной задачи к другой определяет сама задача.

**Вытесняющая многозадачность** – способ реализации многозадачности, при котором момент передачи управления от одной задачи к другой определяет среда выполнения.

# Процессы

**Процесс** – экземпляр выполняющейся программы.

Процесс соответствует программе в целом. Данные программы также являются частью процесса. За исполнение программы отвечают потоки выполнения процесса.

**Поток выполнения** – участок программного кода, который может быть выбран планировщиком для выполнения.

У каждого процесса есть как минимум 1 поток выполнения, который в момент запуска начинает выполнять код программы с начала.

# Изоляция процессов

Каждый процесс имеет *собственные*:

- адресное пространство, в котором размещаются его данные;
  - таблицу дескрипторов;
  - набор блокировок файлов;
  - маску прав создаваемых файлов (см. *man umask*);
  - ID пользователя и группы;
  - набор обработчики сигналов и обработчиков `exit()`;
  - набор потоков выполнения
  - набор ограничений ресурсов;
- и некоторые другие атрибуты.

# Процессы в UNIX

Процессы в UNIX обладают следующими особенностями:

- Процесс однозначно определяется **идентификатором процесса** (process id, **PID**) – уникальным в пределах системы целым неотрицательным числом.
- Процессы порождаются другими процессами путем копирования.
- Процессу известен его процесс-родитель.
- У всего «семейного древа» процессов есть корень – **процесс init (PID=1)**.

# Идентификаторы процесса (пример 1)

Идентификатор процесса и идентификатор его процесса-родителя можно получить вызовами `getpid` и `getppid`.

```
pid_t getpid(); //получить PID
pid_t getppid(); //получить PID родителя
```

Эти вызовы всегда завершаются без ошибок.

*Примечание: в POSIX нет системного вызова, позволяющего вернуть PID дочерних процессов.*

# Создание процесса. Вызов fork (пример 1)

Процесс создается вызовом fork.

```
pid_t fork();
```

Вызов fork создает копию\* вызывающего процесса. Выполнение процесса-потомка начинается с выражения после fork().

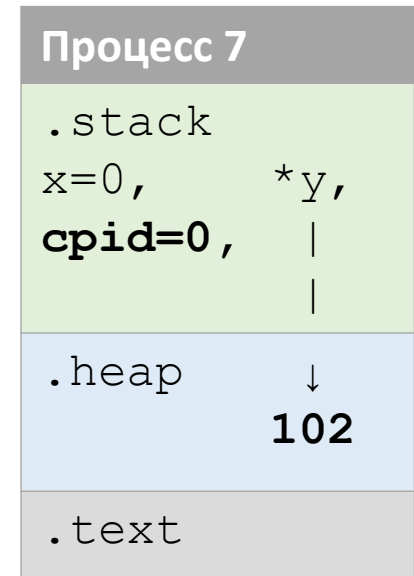
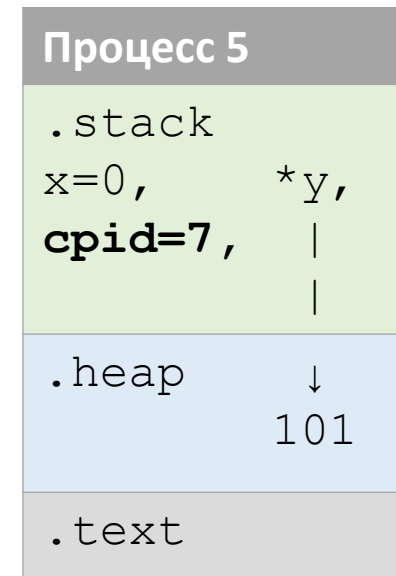
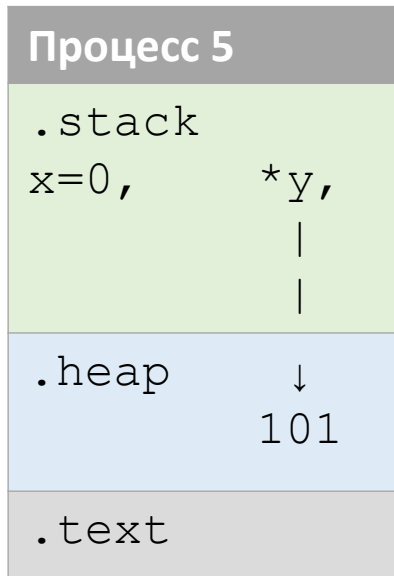
В процессе-родителе вызов возвращает PID созданного процесса либо -1.

В созданном процессе вызов возвращает 0.

\*атрибуты процесса, значения которых не копируются, см. в *man fork*

# Создание процесса. Вызов fork

```
int main() {  
    int x = 0, *y = new int(101);  
    pid_t cpid = fork();  
    if(cpid == 0){*y = 102; /*child code*/ }  
    else        {/*parent code*/}  
}
```





# Завершение работы процесса

Процесс завершается нормально, если:

- завершено выполнение функции `main()`;
- завершено выполнение всех потоков процесса;
- вызвана функция `exit()` или системный вызов `_exit`;

При нормальном завершении процесса ОС сохраняет его **код завершения** – результат `main()` или значение, переданное в `exit()`. По общему соглашению, любой код завершения  $\neq 0$  означает завершение работы в результате ошибки.

Процесс завершается не нормально, если получен сигнал, диспозиция которого – уничтожение процесса.

# Завершение процесса<sub>(пример 2)</sub>

Завершить выполнение процесса можно функцией `exit()`, вызовом `_exit()`.или функцией `abort()`.

```
void exit(int status); /*нормальное завершение*/  
void _exit(int status); /*нормальное завершение*/  
void abort();          /*аварийное завершение*/
```

Функция `exit()` позволяет выполнить дополнительные действия при завершении процесса.

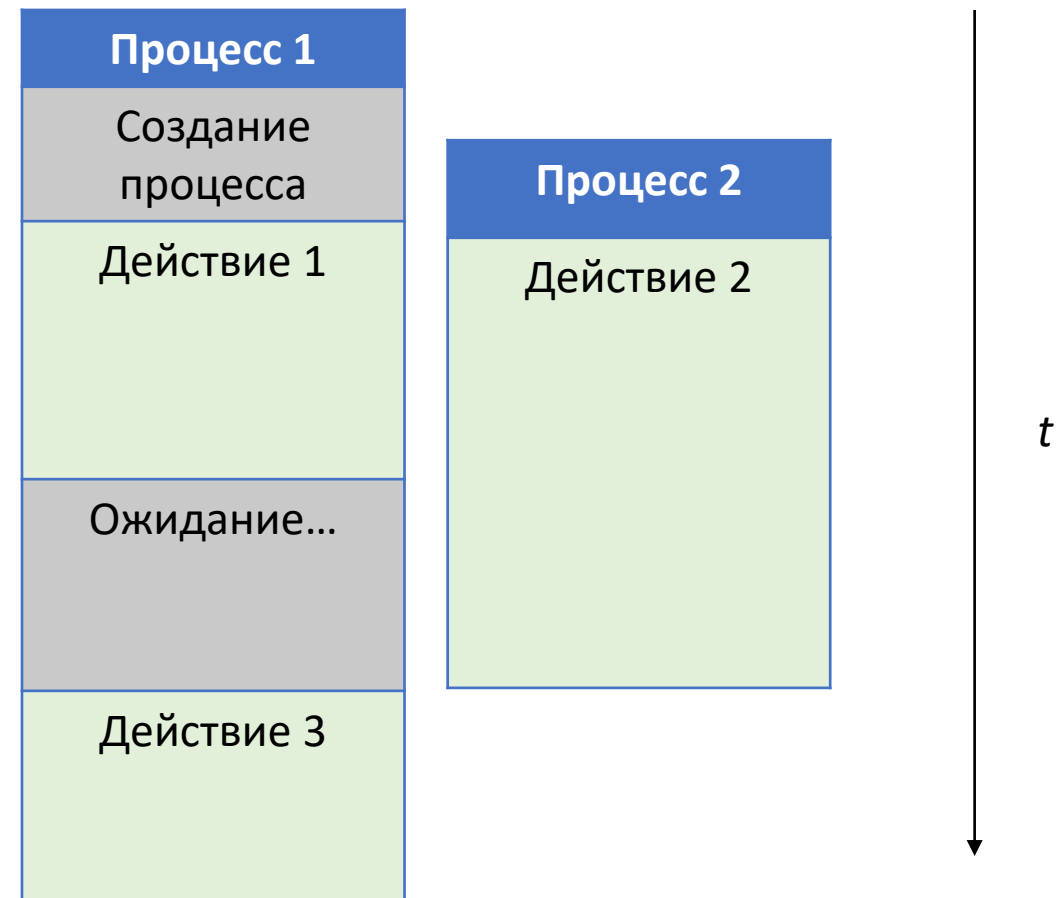
Регистрация функции, которая должна выполняться при завершении работы процесса, осуществляется с помощью функции `atexit()`.

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

# Ожидание завершения дочернего процесса

Часто дочерний процесс создается, чтобы параллельно выполнить некоторое необходимое действие.

Следовательно, нужно дождаться завершения процесса для продолжения работы.



# Вызов wait

Подождать завершения работы одного из дочерних процессов можно вызовом `wait()`.

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

Вызов приостанавливает текущий поток до тех пор, пока не завершится *любой* из дочерних процессов, после чего возвращает PID завершившегося процесса.

Если параметр **wstatus** не равен NULL, по данному указателю записывается информация о завершении процесса.

# Вызов `waitpid` (пример 3)

Вызов `waitpid()` позволяет дождаться изменения состояния конкретного дочернего процесса.

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

Параметры:

`pid` – идентификатор ожидаемого процесса (*man waitpid*);  
`wstatus` – указатель для сохранения информации о завершении процесса;  
`options` – дополнительные флаги (`WNOHANG`).

Если в `pid` передать 0, то вызов будет ожидать любого из дочерних процессов.

Флаг `WNOHANG` позволяет просто проверить, завершился ли данный процесс. Если процесс не завершился, `waitpid()` вернет 0.

# Параметр wstatus

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

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

Для определения конкретной причины завершения процесса используется ряд макросов (возвращают 1 или 0):

- `WIFEXITED(status)` – процесс завершился нормально;
- `WIFSIGNALED(status)` – процесс «убит» сигналом (нет кода завершения);

Если процесс завершился нормально, код завершения можно получить макросом `WEXITSTATUS(status)`.

Номер сигнала, убившего процесс, можно получить макросом `WTERMSIG(status)`

# Таблица процессов. Процессы-зомби

Каждому процессу соответствует запись в системной **таблице процессов**. Данная таблица имеет ограниченный размер.

При завершении работы процесса, запись в таблице не удаляется сразу. В записи продолжает храниться код завершения процесса.

Окончательно запись удаляется из таблицы, когда процесс-родитель забирает код завершения вызовом `wait/waitpid()`.

Процессы, которые завершили выполнение, но о которых еще есть запись в системной таблице процессов, называются **процессами-зомби**.

Слишком большое количество зомби может привести к тому, что системная таблица заполнится, и создавать новые процессы не получится.

*См. также: fork-бомба*

# Процессы-сироты (пример 4)

Если во время работы дочернего процесса завершается его процесс-родитель, дочерний процесс становится **процессом-сиротой**.

Поскольку у каждого процесса должен быть процесс-родитель, процессы-сироты немедленно «усыновляются» одним из процессов-предков\*.

Изначально, все процессы-сироты усыновлялись процессом `init`.

Процесс **`init`** – первый процесс, запускаемый при старте ОС. **PID `init` равен 1**.

Процесс `init` написан так, что всегда забирает код завершения своих дочерних процессов -> `init` предотвращает появление процессов-зомби.

*\* Предок должен явно изъяснить желание усыновлять потомков, см. `man 2 prctl` (секция `PR_SET_CHILD_SUBREAPER`)*



# Структура адресного пространства

Адресное пространство процесса разделено на сегменты, каждый из которых хранит свой тип данных.

- **.text** – код программы.
- **.data** – глобальные переменные.
- **.rodata** – константы.
- **.bss** – неинициализированные глобальные переменные.
- **.heap** – куча (динамическая память).
- **.stack** - программный стек главного потока.

Все сегменты, за исключением **.heap** и **.stack**, копируются из исполняемого файла программы.

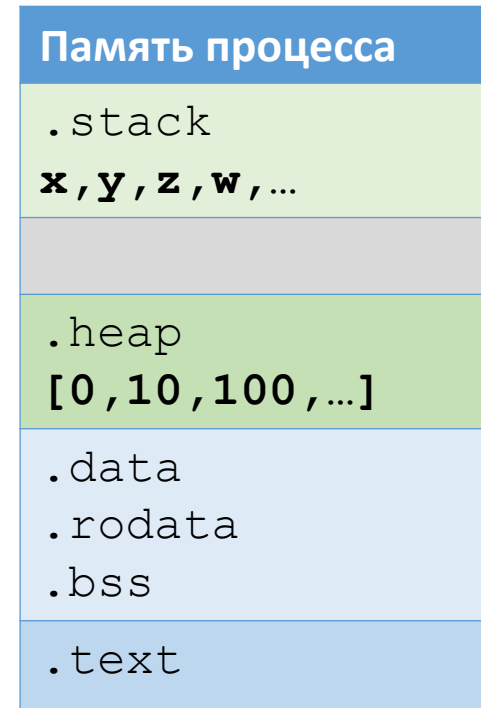
Память процесса
<b>.stack</b> x, y, z, w, ...
<b>.heap</b> [0, 10, 100, ...]
<b>.data</b> <b>.rodata</b> <b>.bss</b>
<b>.text</b>

# Запуск программ в UNIX

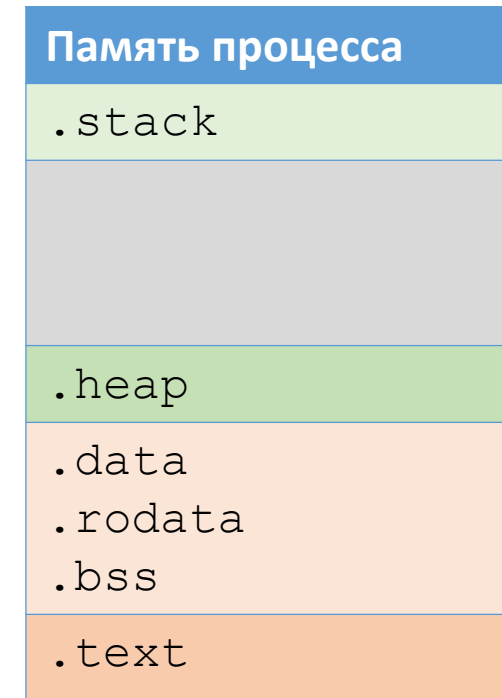
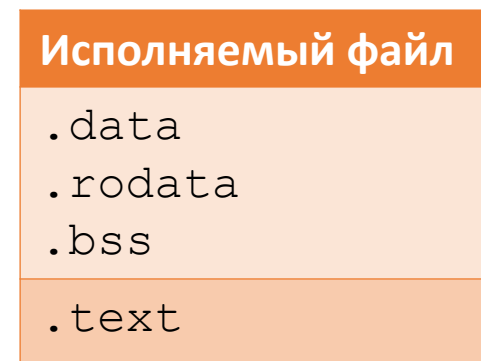
В UNIX новые процессы новый процесс является копией старого.

Для того, чтобы запустить новую программу, используются системные вызовы семейства `exec*()`.

Данные вызовы сбрасывают содержимое сегментов `.stack` и `.heap`, читают новые значения сегментов `.text`, `.data`, `.rodata` и `.bss` из указанного исполняемого файла и начинают выполнение новой программы с начала.



`execve()`



# Запуск программ в UNIX

В момент запуска программы:

- Уничтожается все старое адресное пространство процесса без вызова деструкторов объектов.
- Сбрасываются обработчики сигналов и функции, зарегистрированные `atexit()`.
- Закрываются все открытые очереди, семафоры, объекты разделяемой памяти, каталоги.
- Закрываются все дескрипторы с установленным флагом `FD_CLOEXEC` (*man 2fcntl*).
- Если у исполняемого файла установлен флаг `set-user-id/set-group-id`, производится установка эффективного User-ID/Group-ID процесса.

Обычные открытые файлы не закрываются. Все маски и ограничения сохраняются.

# Вызов `execve` (пример 5)

Для запуска программы из исполняемого файла используется вызов `execve()`.

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

Параметры:

`filename` - путь к исполняемому файлу;

`argv` - аргументы программы;

`envp` - переменные окружения (строки вида *"name=value"*).

Последним элементом массивов `argv` и `envp` должен быть `NULL`.

По общему правилу, `argv[0]` представляет имя запущенной программы. Проще всего в виде `argv[0]` передавать `filename`.

В качестве `envp` проще всего передать переменную `environ`.

# Скрипты и exesve<sub>(пример 5)</sub>

**Скрипт** (сценарий) – текстовый файл, описывающий последовательность действий, исполняемую впоследствии интерпретатором скрипта.

Для того, чтобы скрипт можно было выполнять вызовом `exesve()`, первая строка скрипта должна иметь вид

*#! <команда запуска интерпретатора скрипта>*

При этом соответствующая команда запуска будет иметь вид

*<команда запуска интерпретатора> <путь к скрипту> <аргументы из exesve>*

*Примечание: файл скрипта должен иметь разрешение на выполнение*

# Разделяемые библиотеки

**Разделяемые библиотеки** – исполняемые файлы специального типа, хранящие функции, используемые другими исполняемыми файлами, и загружаемые в ОЗУ во время работы программы.

Обычно разделяемые библиотеки загружаются автоматически в процессе динамической компоновки. Порядок поиска библиотек:

1. Каталоги в переменной окружения `LD_LIBRARY_PATH`;
2. Каталоги в секции `DT_RUNPATH` исполняемого файла;
3. Системный кэш, формируемый системной утилитой *Ldconfig*;
4. Каталоги */lib[64]*, */usr/lib[64]*

Посмотреть список разделяемых библиотек, загружаемых программой, можно утилитой *Ldd*.

# Ручная загрузка библиотек (пример 6)

Загрузить разделяемую библиотеку вручную можно функцией `dlopen()`.

```
void* dlopen(const char *filename, int flags);
```

Параметры:

`filename` – путь к файлу библиотеки;

`flags` – флаги (RTLD\_NOW или RTLD\_LAZY, RTLD\_GLOBAL, и др.).

Вызов возвращает дескриптор библиотеки.

Если в `filename` передать `NULL`, то будет возвращен дескриптор, связанный с таблицей символов всей программы (содержит символы исполняемого файла и всех автоматически загруженных библиотек).

Закрыть дескриптор библиотеки можно вызовом `int dlclose(void *handle);`

Причины ошибок нужно узнавать через `char* dlerror()`, не через `errno`.

# Поиск символов (пример 6)

Получить адрес символа по его имени можно функцией `dlsym()`.

```
void* dlsym(void *handle, const char *symbol);
```

Параметры:

`handle` – дескриптор библиотеки;  
`symbol` – имя символа.

Функция возвращает указатель на символ или NULL.

*Примечание: становится известен только адрес символа, но не его тип. Неизвестно даже, функция это, или переменная.*

Использование ручной загрузки библиотек применяется при построении расширяемых приложений. Обычно в таком случае пользователь может написать библиотеку с определенными функциями и поместить ее по определенному пути. Программа загрузит библиотеку и будет использовать функции из нее.



# Процесс как единица управления ресурсами

Процесс является единицей учета ресурсов, т.к. ОС закрепляет ресурсы за процессом. Учитываются:

- общий объем потребляемой памяти;
- объем кучи и стека;
- общее процессорное время;
- количество открытых файлов;
- размер открытого файла (открытый файл не может «вырасти» за этот предел);
- и т.д. (см. *man 2 setrlimit*)

# Ограничение потребления ресурсов

Для каждого ресурса устанавливаются **мягкий** и **жесткий** пределы потребления.

- при достижении мягкого предела дальнейшее выделение ресурса не производится;
- мягкий предел может быть изменен самой программой, но не может быть выше жесткого предела;
- жесткий предел может быть изменен обычной программой только в сторону уменьшения;
- программы, запущенные от имени суперпользователя, могут менять жесткий предел в сторону увеличения.

# Ограничение потребления ресурсов (пример 7)

Для работы с пределами используются вызовы `getrlimit/setrlimit()`.

```
struct rlimit {  
    rlim_t rlim_cur; /* мягкий предел или RLIM_INFINITY */  
    rlim_t rlim_max; /* жесткий предел или RLIM_INFINITY */  
};
```

```
int getrlimit(int resource, struct rlimit *rlim);
```

```
int setrlimit(int resource, const struct rlimit *rlim);
```

Параметры:

`resource` – константа-тип ресурса (`RLIMIT_AS`, `RLIMIT_NOFILE` и др.),  
`rlim` – указатель на значения предела.

# Каталог `/proc` (пример 8)

Каждый процесс в ОС на ядре Linux имеет отображение в файловой системе в виде каталога `/proc/<PID>`, где `<PID>` - идентификатор процесса.

- `/proc/<PID>/status` – общие сведения о процессе;
- `/proc/<PID>/cwd` – символьная ссылка на рабочий каталог;
- `/proc/<PID>/exe` – символьная ссылка на исполняемый файл;
- `/proc/<PID>/cmdline` – аргументы программы;
- `/proc/<PID>/environ` – переменные среды;
- `/proc/<PID>/limits` – пределы;
- `/proc/<PID>/maps` – карта адресного пространства;
- `/proc/<PID>/mem` – виртуальная память процесса;
- `/proc/<PID>/fd` – каталог с файловыми дескрипторами;
- `/proc/<PID>/task` – каталог с потоками;

# Каталог /proc

- */proc/self* – ссылка на каталог текущего процесса
- */proc/thread-self* – ссылка на каталог текущего потока
- */proc/version* – версия ОС
- */proc/cpuinfo* – информация о ЦП
- */proc/meminfo* – информация об использовании ОЗУ
- */proc/filesystems* – доступные для монтирования файловые системы
- */proc/sys* – каталог с системной информацией
- */proc/sys/kernel* – каталог ядра ОС