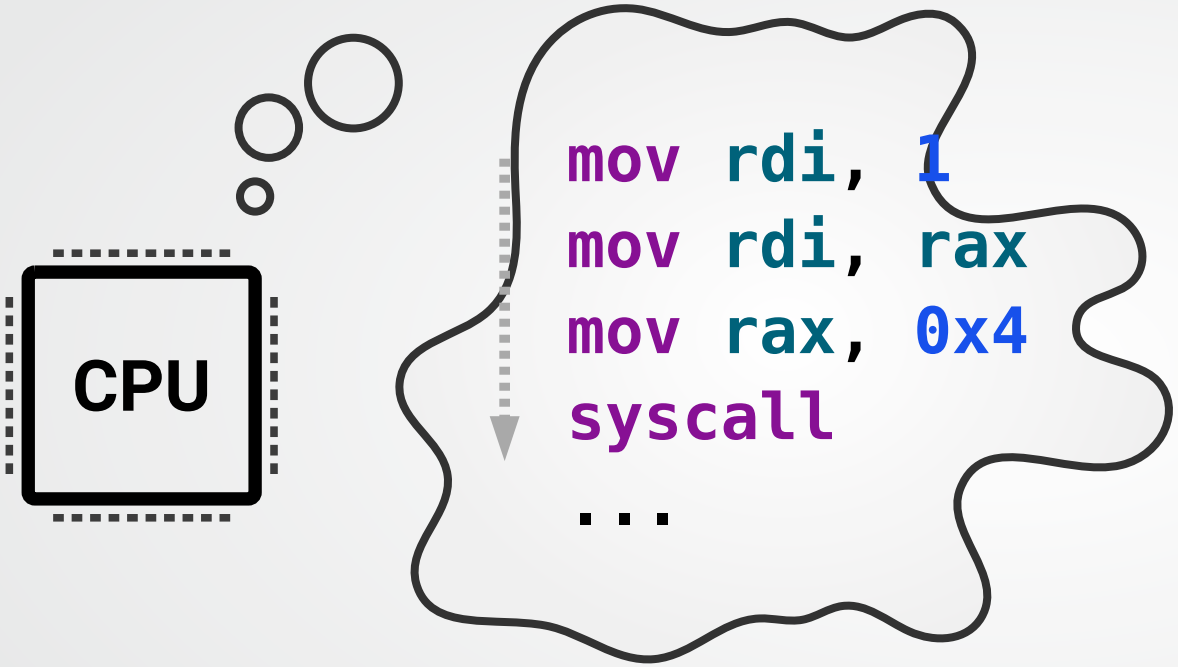
Процессы и потоки

АКОС, МФТИ

24 октября, 2024

Программа 1:



Программа 2:

```
mov r10, 1
push r10
mov rdi, r10
call 0x10c90
```

Как запустить вторую программу?

Можно использовать несколько ядер!

```
mov rdi, 1
mov rdi, rax
mov rax, 0x4
syscall
...
```

```
mov r10, 1
push r10
mov rdi, r10
call 0x10c90
...
```

Первое ядро

Работает с программой 1

Второе ядро

Работает с программой 2

Можно использовать несколько ядер!

```
mov rdi, 1
mov rdi, rax
mov rax, 0x4
syscall
...
```

```
mov r10, 1
push r10
mov rdi, r10
call 0x10c90
...
```

Первое ядро

Работает с программой 1

Второе ядро

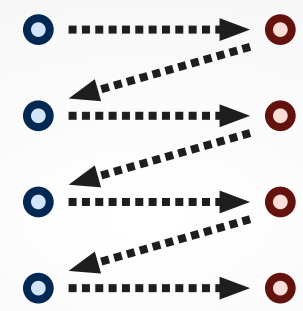
Работает с программой 2

Но ядер мало...

Можно ли выполнять две программы на одном ядре?

```
mov rdi, 1
mov rdi, rax
mov rax, 0x4
syscall
...
```

Чередуя инструкции:



```
mov r10, 1
push r10
mov rdi, r10
call 0x10c90
...
```

Что пойдет не так?

Что пойдет не так?

- Программы будут портить друг другу **регистры**;

Что пойдет не так?

- Программы будут портить друг другу **регистры**;
- …И **память** (вспомним про адресные пространства).

Что пойдет не так?

- Программы будут портить друг другу **регистры**;
- …И **память** (вспомним про адресные пространства).

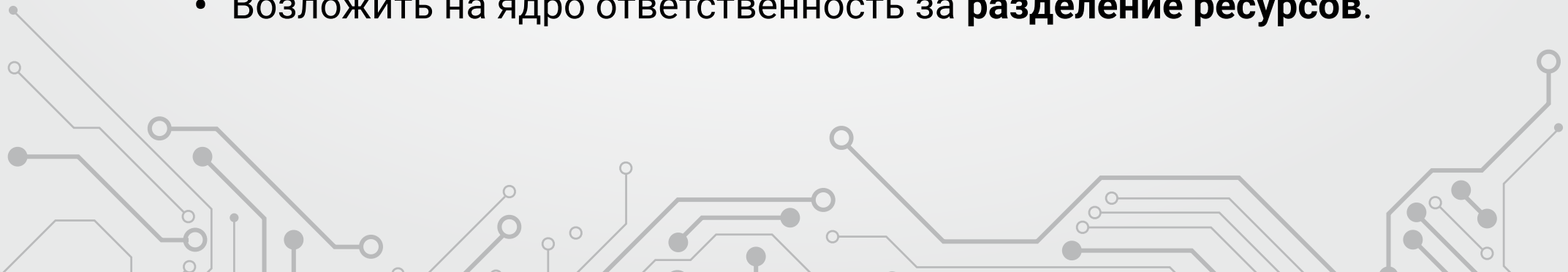
Как это починить?

Что пойдет не так?

- Программы будут портить друг другу **регистры**;
- …И **память** (вспомним про адресные пространства).

Как это починить?

- Возложить на ядро ответственность за **разделение ресурсов**.



Контекст 1

```
mov rdi, 1
mov rsi, rax
```

Код ядра ОС — *Переключение контекста*

Контекст 2

```
mov r10, 1
push r10
```

Код ядра ОС — *Переключение контекста*

Контекст 1

```
mov rax, 0x4
syscall
```

Прерывания по таймеру

60 - 1000 раз / сек.

Переходы в Userspace

Как переключить контекст

1. **Сохранить** в память контекст текущей программы;
2. **Восстановить** из памяти контекст новой программы;
3. Перейти в Userspace и **продолжить исполнение** новой программы.

- Происходит по прерыванию таймера;
- **Позволяет одновременно запустить много программ на одном ядре**.

Переходим к правильным терминам:

Программа → Процесс

- Имеет свой **номер** (PID)
- Имеет свой **контекст выполнения**
- Имеет своё **адресное пространство**
- Имеет свой **набор дескрипторов**
- Знает своего **родителя** и **пользователя**
- *И еще много чего…*

Создаём процессы, как это делали наши отцы

`fork()`

- **Дублирует** текущий процесс, включая указатель на текущую инструкцию;
- **Возвращается дважды**:
  - В созданном процессе возвращает `0`;
  - В родительском процессе возвращает номер (PID) созданного процесса.

`exec(...)`

- **Заменяет** текущий процесс на новый процесс по командной строке;
- Имеет много вариаций: `execl(...)`, `execv(...)`, `execve(...)` и т.д;
- **Наследует настройки** родительского процесса: открытые дескрипторы, переменные окружения, рабочую директорию, маски сигналов и т.д.

`fork()` + `exec()`

```c
if (fork() == 0) {
    // Здесь можно закрыть всё лишнее
    // И заместить себя другим процессом
    execl("/bin/bash", NULL);
}

// Здесь полезная работа родителя

wait(NULL); // Ждём завершения потомка
```

…или как написать свой `$ bash` в 9 строк.

А что, если...

```
1  while (true) {
2    fork();
3  }
```

...и отбежать на безопасное расстояние?

Создаём процессы модно

```
posix_spawn(pid_t* pid, char* file, /* whole lot of arguments */)
```

- Умно-хитро **создаёт новый процесс** по командной строке и исполняемому файлу.
- Даёт **широкий набор настроек** для создаваемого процесса, например:

`posix_spawnattr_setsigmask(...)`	Настройка маски сигналов;
`posix_spawnattr_getpgroup(...)`	Настройка группы процессов;
`posix_spawnattr_setschedparam(...)`	Настройка параметров планирования;
`posix_spawn_file_actions_xxxxxx(...)`	Открытие и закрытие дескрипторов.

- **Быстрее, чем** `fork()` **+** `exec()` .

Дожидаемся процессов

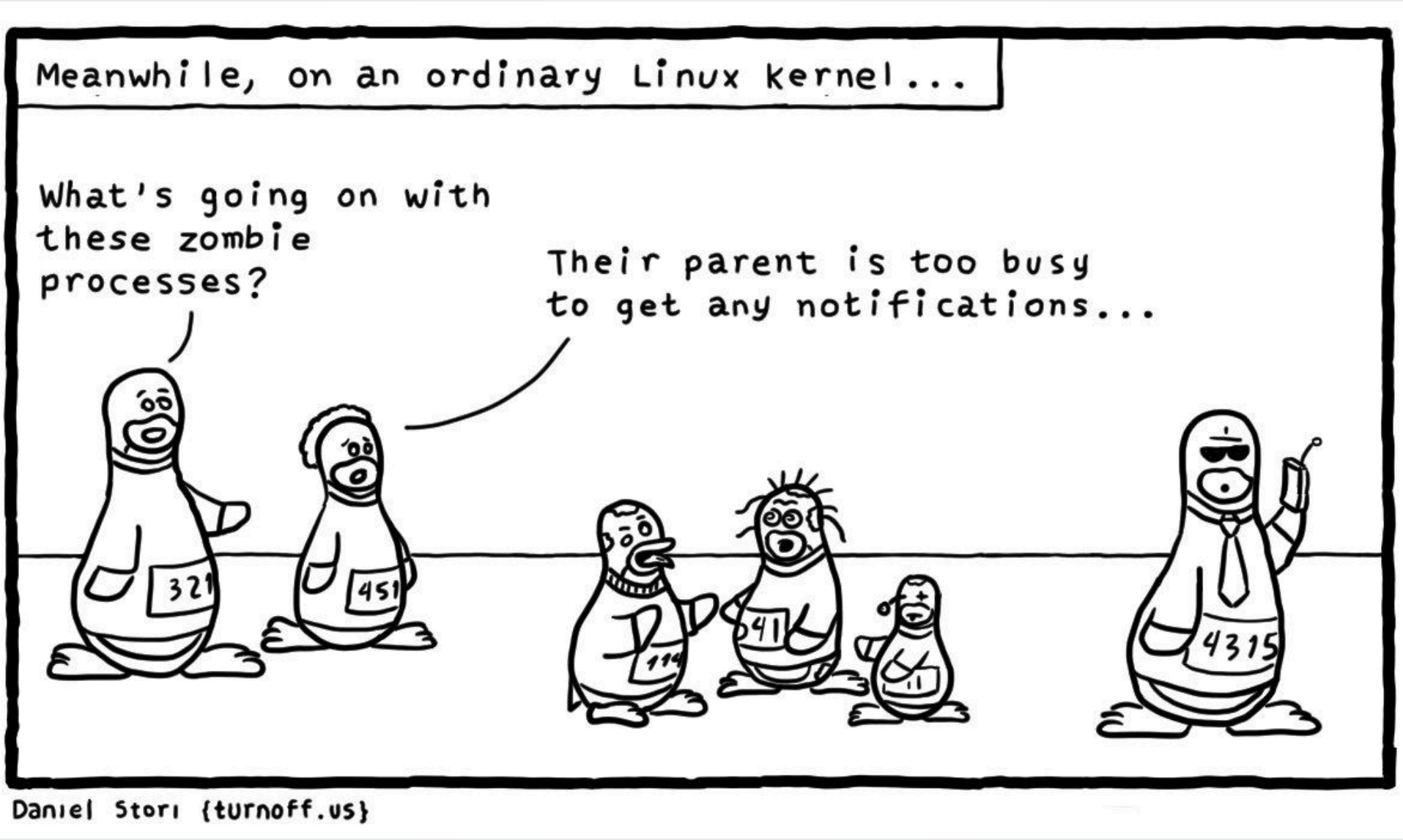
`waitpid(pid_t pid, int* status, int options)`

Ждёт завершения процесса и возвращает статус завершения.

`pid_t pid`	**Процесс**, которого нужно дождаться. `-1` = любой дочерний;
`int* status`	Куда вернуть **статус завершения** процесса. Может быть `NULL`
`int options` :	
`|= WNOHANG`	**Не ждать** дочерний процесс, если он ещё работает;
`|= WUNTRACED`	Сработать на **остановку** процесса (`SIGSTOP`).
`|= WCONTINUED`	Сработать на **возобновление** процесса (`SIGCONT`).

`wait(status)` ≡ `waitpid(-1, status, 0)` . И то, и то под капотом – `wait4(...)`.

⚠️ : **Ждать нужно каждого дочернего процесса**, иначе они станут зомби.


Meanwhile, on an ordinary Linux kernel...
What's going on with these zombie processes?
Their parent is too busy to get any notifications...
321
451
114
541
1
4315
Daniel Stori {turnoff.us}

Статусы `waitpid(...)`

⚠️ : Статус, который передаёт `waitpid(...)` – это **не код завершения процесса.**

Это битовая маска, которую можно декодировать библиотечными функциями:

`WIFEXITED(status)`	Процесс завершился **нормально**;
`WIFSTOPPED(status)`	Процесс **приостановлен**;
`WIFCONTINUED(status)`	Процесс **возобновлен**;
`WIFSIGNALED(status)`	Процесс завершился **сигналом**;
`WEXITSTATUS(status)`	Какой **код завершения** вернул процесс;
`WSTOPSIG(status)`	Какой **сигнал** приостановил процесс;
`WTERMSIG(status)`	Какой **сигнал** завершил процесс;

Как использовать несколько ядер в одном процессе?

Поток

- Имеет свой **номер** (TID)
- Имеет свой **контекст выполнения**
- Имеет свой **стек**

- Является **частью процесса**
- Почти всё **делит с другими потоками**
- **Общее адресное пространство**
- **Общий набор дескрипторов**

Зачем нужны потоки?

Потоки позволяют осуществлять параллельные вычисления внутри процесса.

- Например, для софтверного рендеринга:
  - Разделить картинку на несколько частей;
  - Каждую часть рендерить в своём потоке.

⚠️ : **Потоки и процессы - не одно и то же!**

`pthread`

- **Ваш инструмент** для создания потоков и контроля над ними.
- Подключается через `#include <pthread.h>` и флаг `-pthread` .

```
pthread_create(pthread_t*, pthread_attr_t*, f* function, void* arg);
```

Конструктор структуры `pthread_t`

`pthread_t thread`	**Структура**, которую нужно инициализировать
`const pthread_attr_t* attr`	**Атрибуты** потока
`(void*)(*function)(void*)`	**Entrypoint** потока
`void* arg` :	**Аргумент** для entrypoint

⚠️ : **Деструктора у** `pthread_t` **нет,** только у атрибутов.

- Поток уничтожится сам, когда выполнятся определённые условия.

Атрибуты потока – `pthread_attr_t`

`pthread_attr_init(...)`	**Конструктор**;
`pthread_attr_destroy(...)`	**Деструктор** (да, у атрибутов он есть);
`pthread_attr_setstacksize(...)`	Запросить другой **размер стека**;
`pthread_attr_setguardsize(...)`	Запросить другой **размер guard-секции**;
`pthread_attr_setstack(...)`	Установить **собственный стек**;
`pthread_attr_setaffinity_np(...)`	Настроить набор процессорных ядер;

И еще много чего…

Если вас устраивают **атрибуты по умолчанию**, можно передать `NULL`

Атрибуты можно освободить сразу после создания потока, либо переиспользовать.

Неявное завершение работы

- **Поток завершил свою работу**, вернувшись из своей функции.

Явное завершение работы

- **Поток явно завершил работу**: `pthread_exit(...)`
- **Другой поток отменил его**: `pthread_cancel(...)`

Поток освобождается, когда он завершил свою работу и:

- Либо его **дождался** другой поток через `pthread_join(...)`
- Либо его **пометили как отсоединённый** через `pthread_detach(...)`

`pthread_join(pthread_t tid, void **status)`

Ждет завершения потока и возвращает статус завершения.

`pthread_t tid`	**Поток**, которого нужно дождаться.
`void** status`	Куда вернуть **статус завершения** процесса. Может быть `NULL`

- `pthread_join(...)` **- аналог** `waitpid(...)` **для потоков.**
- **Если поток был отменён**, то вместо кода возврата вернётся `PTHREAD_CANCELED`.
- Пока вы не вызовете `pthread_join(...)`, поток **не сможет освободиться**.

⚠️ Осторожно, UB!

- **После** `pthread_join(...)` **поток уже освобождён.**
- Работа с ним - **UB**

`pthread_detach(pthread_t tid)`

Помечает поток как отсоединённый. Ничего не ждёт.

Отсоединенный поток освобождается из памяти **сразу по завершении работы**.

Если проще - `pthread_detach(...)` ≡ `pthread_join(...)` отложенного действия.

Этим можно пользоваться, когда код возврата не нужен.

⚠️ Осторожно, здесь тоже UB!

- **После `pthread_detach(...)` поток может освободиться в любой момент.**
- Работа с ним - **UB**

⚠️ Кругом UB!

Не все стандартные функции любят, когда их используют многопоточно. Например:

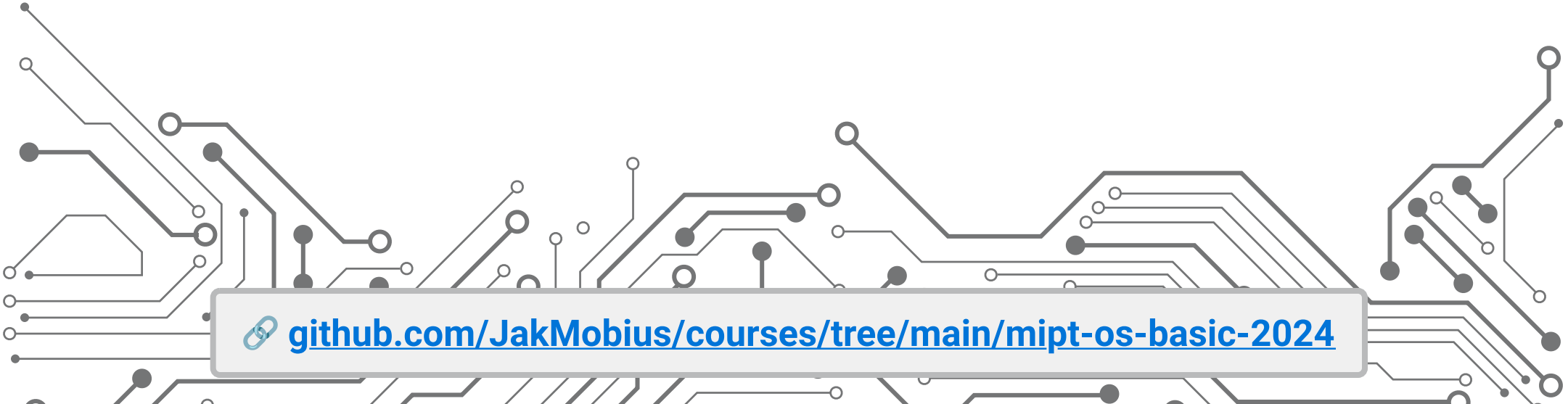
`crypt()`	`ctime()`	`encrypt()`	`dirname()`
`localtime()`	`rand()`	`strerror()`	`getdate()`

- **Функции могут иметь разную толерантность к многопоточности**. Например:
  - Быть безопасными, **но не на первом вызове**;
  - Быть безопасными, **но не на одинаковых объектах**;
  - Быть безопасными, **но только если окружение не меняется**;
  - И так далее.

Об этом подробно сказано в мануалах (man 7 attributes, man 7 pthreads)

Многопоточное программирование кишит UB, но об этом в следующий раз.

Спасибо за внимание!



🔗 github.com/JakMobius/courses/tree/main/mipt-os-basic-2024