Программно-аппаратные платформы Интернета вещей и встраиваемые системы

Лекция 3

МНОГОЗАДАЧНОСТЬ

Зачем нужна многозадачность?

```
int main(void){
    puts("Hello World!");
    while (1){
        task1();
        task2();
        task3();
    }
    return 0;
}
```

- Бесконечный цикл, он же loop() в Arduino
- Очень быстро сталкиваемся с тем, что задания «мешают» друг другу
- При программировании отдельной задачи надо учитывать и остальные
- Сложно использовать готовые библиотеки

Зачем нужна многозадачность?

- Много периодических задач – опрос датчика, отправка данных через радиомодем, протокол обмена по радио со сложными требованиями к времени отправки и приема сообщений
- Можно взять LoRaMAC-Node и долго его дорабатывать

Логика в прерываниях

- «Система, управляемая событиями» - event-driven system
- Логика выполнения программы находится в обработчиках прерываний
- Подход широко применяется при сравнительно несложной и быстрой обработке внешних событий
- Пример – UMDK-RF, UMDK-EMB https://github.com/a-podshivalov/dap42

Атомарные операции

- Чтение-модификация-запись
- Что произойдет, если эта последовательность прервется из другой задачи/прерывания? Что произойдет при одновременном доступе из двух задач?
- `volatile` не поможет
- Два способа обеспечить выполнение последовательности целиком:
  - Запрет прерываний (и смены контекста из планировщика)
  - Операции неблокирующего эксклюзивного доступа к памяти (LDREX/STREX для ARM)

Виды многозадачности

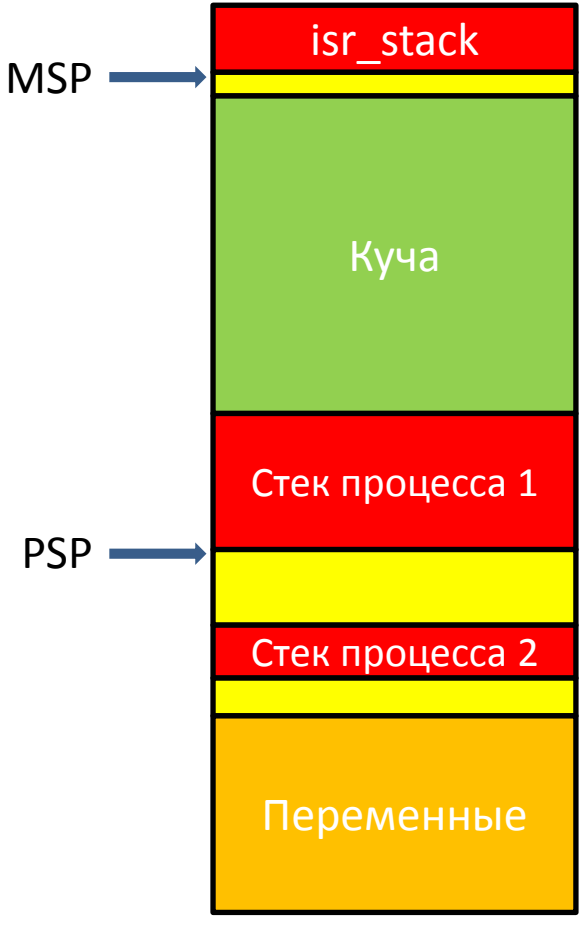
- Добавляем **планировщик задач**
- Вытесняющая многозадачность
  - Планировщик вызывается по прерыванию таймера
  - «Повисшая» задача не останавливает остальные
  - Повышенный расход энергии
- Кооперативная (невытесняющая, tickless) многозадачность
  - Планировщик вызывается при переходе очередной задачи в режим ожидания
  - «Повисшая» задача останавливает работу системы
  - Когда все задачи находятся в режиме ожидания – можно включить энергосберегающий режим

Аппаратная поддержка многозадачности в Cortex-M

- Таймер SYSTICK (24 бита, тактирование от основного тактового генератора)
- Два указателя стека

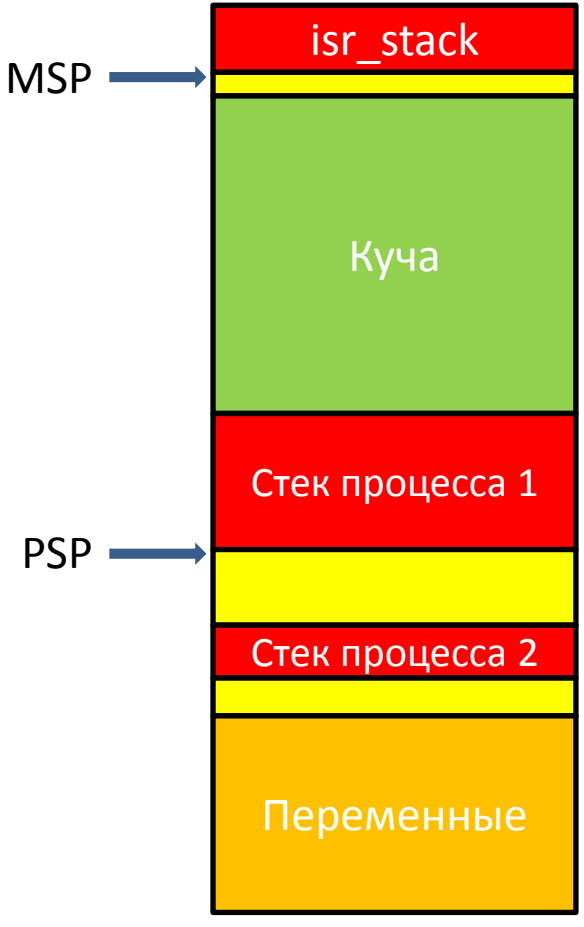
  MSP – main stack pointer

  PSP – process stack pointer
- Два режима работы – Thread Mode, Handler Mode
- В большинстве ОС стек процесса – это просто достаточно большой массив char



Смена контекста (на примере RIOT)

- Выполняется из Handler Mode, то есть из прерывания (PendSV или любого другого, из функции `cortexm_isr_end`)
- Сохраняем регистры и статус процессора для текущего потока (в его стеке), вызываем планировщик
- Загружаем их из обработчика прерывания SVC (Supervisor Call)



`cpu/cortexm_common/thread_arch.c`

Многозадачность в RIOT

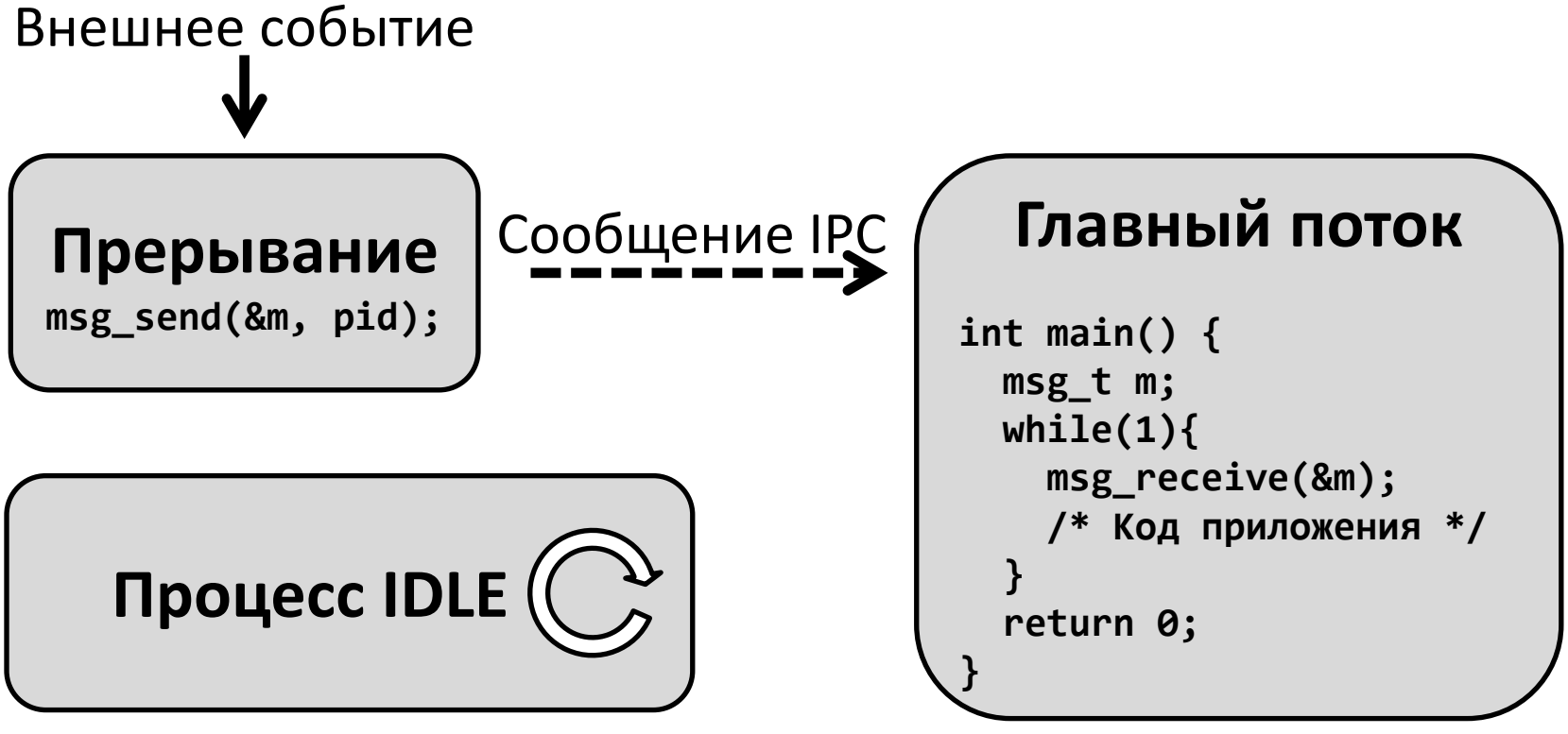
- Tickless-планировщик, сложность O(1)
- 16 приоритетов процессов (процесс/поток/задача – это одно и то же)
  - 0 – высший приоритет, 15 – низший (для процесса IDLE)
  - Процесс main имеет приоритет 7
- Прерывание может прервать работу любого процесса (все прерывания имеют одинаковый приоритет, вложенные прерывания не поддерживаются)
- По завершении прерывания может быть вызван планировщик, который выберет незаблокированный процесс с наибольшим приоритетом

Создание процесса

```
static char my_stack[THREAD_STACKSIZE_DEFAULT];
void* my_thread(void* arg){
    while(1){
        lptimer_sleep((int) arg);
        gpio_toggle(LED0_PIN);
    }
}

int main(void) {
    thread_create(my_stack, sizeof(my_stack),
                  THREAD_PRIORITY_MAIN-1,
                  THREAD_CREATE_STACKTEST,
                  my_thread,
                  (void*) 500,
                  "My own thread");
    return 0;
}
```

Межпроцессное взаимодействие: сообщения



Назначение приоритетов

- Как задать приоритеты процессов?
- Rate Monotonic Scheduler (Liu & Layland, 1973)
  - Пусть имеется n процессов («событий»), C_i - время обработки i-го события, T_i - период поступления событий
  - Наибольший приоритет назначается процессу с наименьшим T_i
  - Если

$$U = \sum_{i=0}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \underset{n\to\infty}{\longrightarrow} \ln 2 \approx 0{,}69$$

то все процессы успеют обработать свои события

Межпроцессное взаимодействие: сообщения

- У каждого потока есть «почтовый ящик» – по умолчанию на 1 сообщение (`msg_t`), можно увеличить функцией `msg_init_queue`
- Отправка сообщений:
  - `msg_send` – блокирующая (за исключением прерываний)
  - `msg_try_send` – неблокирующая
- Получение сообщений:
  - `msg_receive`, `msg_try_receive`
  - `xtimer_msg_receive_timeout`, `lptimer_msg_receive_timeout`

Межпроцессное взаимодействие: mutex

- **Mut**ual **Ex**clusive
- Две основных функции:
  - `mutex_lock` (`mutex_trylock` – неблокирующая версия); нельзя использовать из прерываний
  - `mutex_unlock`, можно использовать из прерываний и других потоков
- Если поток попытался захватить уже занятый mutex, то он блокируется до тех пор, пока mutex не будет освобожден
- “The plural of mutex is deadlock”

Deadlock

Инверсия приоритетов

- Если низкоприоритетный поток захватил mutex, то потоки с высоким приоритетом вынуждены его ждать
- Еще хуже – пока все ждут, начал работу поток со «средним» приоритетом, не претендующий на этот mutex – пока он не завершится, mutex не будет разблокирован

Практика применения

- Mutex’ы используются для управления доступом к периферии (SPI, I2C и тому подобные интерфейсы)
- Этими интерфейсами не рекомендуется пользоваться из прерываний
- Обработчик прерывания – максимально простая функция, отправляющая сообщение потоку (слайд 12)

Пример: драйвер ST95

- ST95HF – приемопередатчик NFC
  - `drivers/st95/st95.c`, `drivers/include/st95.h`
  - `unwired-modules/umdk-st95/umdk-st95.c`
- В функции `umdk_st95_init` инициализируется датчик, создается поток для обработки прерывания от датчика (`radio_pid`)
- Непосредственно прерывание обрабатывается в драйвере (`st95.c`), функции `_st95_uart_rx`, `_st95_spi_rx`
- Из обработчика прерывания передается сообщение потоку:

```
static void wake_up_cb(void * arg){
    (void) arg;
    msg_try_send(&msg_wu, radio_pid);
}
```

Пример: netdev, драйвер SX1276

- SX1276 – модем LoRa
  - `drivers/sx1276/`
- Реализован интерфейс netdev:

```
const netdev_driver_t sx127x_driver = {
    .send = _send,
    .recv = _recv,
    .init = _init,
    .isr = _isr,
    .get = _get,
    .set = _set,
};
```

Пример: netdev, драйвер SX1276

- При использовании драйвера создаем дополнительный поток для обработки прерываний (в нем же можно производить полезную работу)
- pid потока указываем в поле `event_callback` структуры `netdev`
- При получении потоком сообщения с типом `MSG_TYPE_ISR` вызываем функцию `isr()` из драйвера

Пример: netdev, драйвер SX1276

```
void *isr_thread(void *arg){
    (void)arg;
    static msg_t _msg_q[SX127X_LORA_MSG_QUEUE];
    msg_init_queue(_msg_q, SX127X_LORA_MSG_QUEUE);
    while (1) {
        msg_t msg;
        msg_receive(&msg);
        if (msg.type == MSG_TYPE_ISR) {
            netdev_t *dev = msg.content.ptr;
            dev->driver->isr(dev);
        } else {
            puts("[LoRa] unexpected msg type");
        }
    }
}
```