Аппаратное обеспечение IoT/CPS
Лекция 3

А. А. Подшивалов
apodshivalov@miem.hse.ru

Операционные системы реального времени

- **Расширенная машина:** унифицированный HAL, переносимость между различными семействами и архитектурами МК
- **Менеджер ресурсов:** управление процессорным временем, доступом к памяти и периферии
- **Реальное время:** гарантированное время реакции на события

Управление ресурсами

Менеджер ресурсов: этот слой состоит из мощных функциональных модулей, реализующих стратегические задачи по управлению основными ресурсами вычислительной системы

Многозадачность и IoT

Несколько независимых «задач»:

- Периодический (10 раз в секунду) опрос геркона
- Опрос магнетометра
- Отправка данных по радио (с учетом требований протокола передачи данных)
- Многие задачи представляются в виде конечного автомата

Пробуем обойтись без ОС

Без операционной системы: «суперцикл»

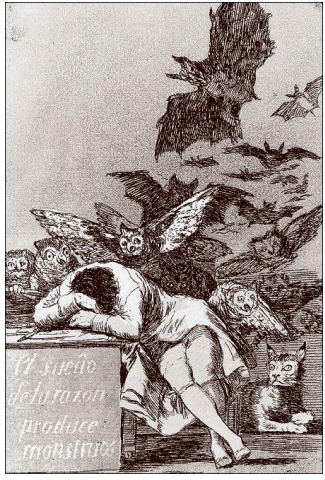
```
int main(void){
    puts("Hello World!");
    while (1){
        task1();
        task2();
        task3();
    }
    return 0;
}
```

- Выполняем все задачи последовательно, внутри бесконечного цикла
- При программировании каждой из задач необходимо учитывать и другие
- Сложно использовать готовые библиотеки
- «Операционная система» POSIT для PIC
- `loop()` в Arduino

Пример: OpenBCI

```
void loop() {
    static byte samplesCounter = 0;
    if(is_streaming){
        while(!(OBCI.isDataAvailable())){}
        OBCI.updateChannelData();
        if(fileIsOpen) {
            writeDataToSDcard(samplesCounter);
        }
        samplesCounter++;
        if(is_running){
            OBCI.sendChannelData();
        }
    }
    eventSerial();
}
```

Сон разума рождает чудовищ



Сон разума рождает чудовищ (Betaflight/Cleanflight)

```
void scheduler(void) {
    // skipped some code
    if (gyroEnabled) {
        // Realtime gyro/filtering/PID tasks get complete priority
        task_t *gyroTask = getTask(TASK_GYRO);
        const timeUs_t gyroExecuteTimeUs = getPeriodCalculationBasis(gyroTask) \
                                                                + gyroTask->desiredPeriodUs;
        gyroTaskDelayUs = cmpTimeUs(gyroExecuteTimeUs, currentTimeUs);
        if (cmpTimeUs(currentTimeUs, gyroExecuteTimeUs) >= 0) {
            taskExecutionTimeUs = schedulerExecuteTask(gyroTask, currentTimeUs);
            if (gyroFilterReady()) {
                taskExecutionTimeUs += schedulerExecuteTask(getTask(TASK_FILTER), currentTimeUs);
            }
            if (pidLoopReady()) {
                taskExecutionTimeUs += schedulerExecuteTask(getTask(TASK_PID), currentTimeUs);
            }
            currentTimeUs = micros();
            realtimeTaskRan = true;
        }
    }
    if (!gyroEnabled || realtimeTaskRan || (gyroTaskDelayUs > GYRO_TASK_GUARD_INTERVAL_US)) {
        // Update task dynamic priorities
        for (task_t *task = queueFirst(); task != NULL; task = queueNext()) {
```

Event-driven системы

Давайте активнее использовать прерывания!

- Реагируем на внешние события
- Логика работы программы находится в обработчиках прерываний
- Обрабатываем события максимально быстро, не допуская блокировок
- Пример: UMDK-RF, UMDK-ENERGYMON

https://github.com/a-podshivalov/dap42

Атомарность операций

- Последовательность чтение—модификация—запись
- Что произойдет при нарушении этой последовательности двумя задачами?
- `volatile` не поможет
- Два способа обеспечить выполнение последовательности операций целиком:
  - Запрет прерываний (критическая секция)
  - Неблокирующий эксклюзивный доступ к памяти (`LDREX`/`STREX` для ARM Cortex-M3 и выше)

Пример: язык nesC и TinyOS

```
module SurgeM {
    provides interface StdControl;
    uses interface ADC;
    uses interface Timer;
    uses interface Send;
}
implementation {
    uint16_t sensorReading;
    command result_t StdControl.init() {
        return call Timer.start(TIMER_REPEAT, 1000);
    }
    event result_t Timer.fired() {
        call ADC.getData();
        return SUCCESS;
    }
    event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        ... send message with data in it ...
        return SUCCESS;
    }
    ...
}
```

Пример: язык nesC и TinyOS

- Операционная система для устройств IoT на базе простых 8-битных МК (1-4 кБ RAM, 8-128 кБ ROM)
- Расширение языка Си
- Задачи (tasks) выполняются в последовательности, определяемой планировщиком (синхронный код)
- События (events) происходят в произвольные моменты времени и вытесняют задачи (асинхронный код, соответствуют прерываниям от аппаратуры)
- Блокирующие операции отсутствуют, если выполнение операции связано с ожиданием — то окончанию ее выполнения соответствует событие
- Есть средства статического анализа кода

Coroutines: почти настоящая многозадачность

Небольшое отступление: Duff's device

```
send(short *to, short *from, int count){
    int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
        } while (--n > 0);
    }
}
```

Coroutines/protothreads на Си

```
struct pt { int16_t lc; };
#define PT_BEGIN(pt)          switch((pt)->lc) { case 0:
#define PT_WAIT_UNTIL(pt, c)  pt->lc = __LINE__; case __LINE__: if(!(c)) return 0
#define PT_END(pt)            } (pt)->lc = 0; return 2
#define PT_INIT(pt)           (pt)->lc = 0

int example(struct pt *pt) {
    PT_BEGIN(pt);
    while(1) {
        if(initiate_io()) {
            timer_start(&timer);
            PT_WAIT_UNTIL(pt, io_completed() || timer_expired(&timer));
            read_data();
        }
    }
    PT_END(pt);
}
```

Пример: операционная система Contiki

- Кооперативная многозадачность на основе coroutines/protothreads
- Для сохранения состояния «потока» нужно только 2 байта
- Внутри «потоков» нельзя использовать `switch`, все локальные переменные надо объявлять с модификатором `static`
- Несколько эталонных реализаций сетевых протоколов для IoT

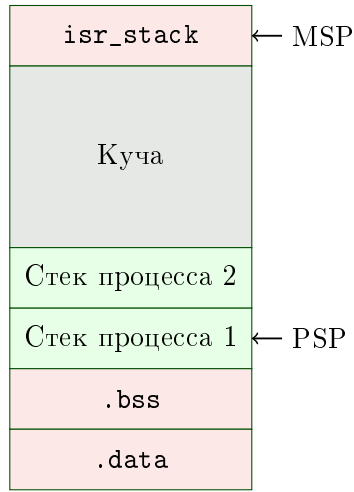
Системы с вытесняющей многозадачностью

Вытесняющая многозадачность

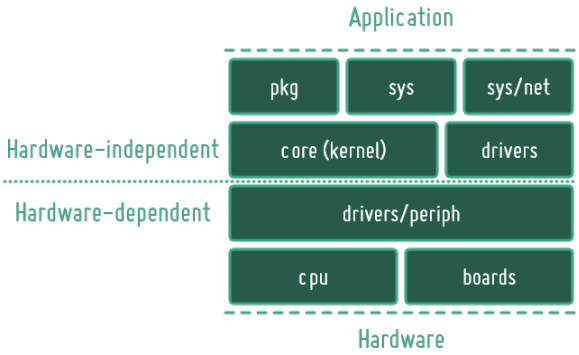
- Основные принципы — проект μITRON (с 1984 года)
- Процессы (потоки, задачи) полностью независимы
- Каждый процесс имеет свой собственный стек вызовов
- Переключение между процессами осуществляет планировщик
- Межпроцессное взаимодействие — очереди сообщений, семафоры, mutex'ы
- Вызов планировщика осуществляется либо по таймеру, либо когда это разрешит выполняющийся процесс (кооперативная и tickless многозадачность)

Аппаратная поддержка многозадачности в Cortex-M

- Два режима работы — Handler Mode, Thread Mode
- Два указателя стека
  - MSP — Main Stack Pointer
  - PSP — Process Stack Pointer
- Прерывания PendSV, SVC
- Таймер SYSTICK (24 бита, тактирование от основного тактового генератора)



Операционная система Riot

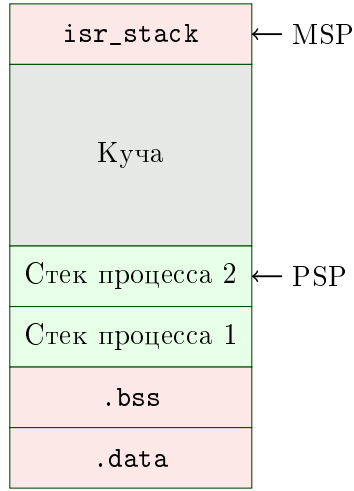


- tickless-планировщик, $O(1)$
- 16 приоритетов процессов, 0 — высший приоритет, 15 — низший (процесс `idle`), процесс `main` имеет приоритет 7
- Прерывание может прервать работу любого процесса, все прерывания имеют одинаковый приоритет (возможно, кроме PendSV с наименьшим приоритетом)
- По завершении прерывания при необходимости вызывается планировщик

Переключение контекста в Riot на Cortex-M

- Функция `thread_yield_higher()` вызывается либо потоком, желающим «уступить», либо из прерывания при необходимости
- Вызывается прерывание PendSV, в стек текущего потока сохраняется его **контекст**
- С отключенными прерываниями вызывается планировщик (функция `sched_run()`)
- Восстанавливается контекст выбранного планировщиком потока

`cpu/cortexm_common/thread_arch.c`



Создание процесса

```
static char my_stack[THREAD_STACKSIZE_DEFAULT];
void* my_thread(void* arg){
    while(1){
        xtimer_msleep((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");
    // do something
    return 0;
}
```

Назначение приоритетов

Rate Monotonic Scheduler (Liu & Layland, 1973):

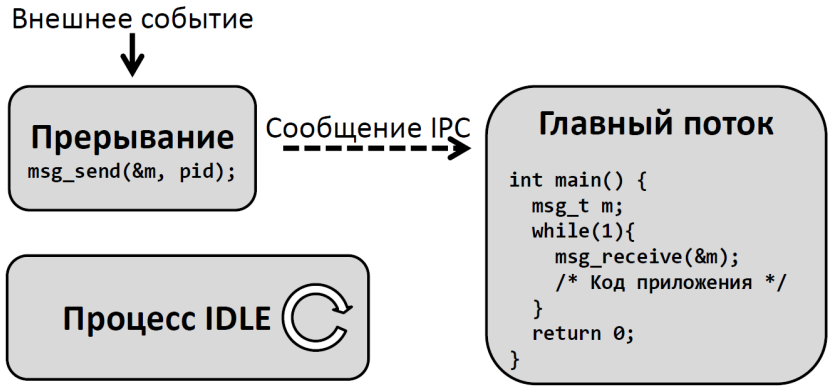
- Пусть имеется n процессов («событий»), C_i — время обработки i-го события, T_i — период поступления событий (или требуемое время реакции)
- Наибольший приоритет назначается процессу с наименьшим T_i
- Если

$$U = \sum_{i=0}^{n} \frac{C_i}{T_i} \leqslant n\left(2^{\frac{1}{n}} - 1\right) \xrightarrow[n\to\infty]{} \ln 2 \approx 0,69,$$

  то все процессы успеют обработать свои события

Примитивы межпроцессного взаимодействия

Сообщения


Внешнее событие
Прерывание
msg_send(&m, pid);
Сообщение IPC
Главный поток
int main() {
msg_t m;
while(1){
msg_receive(&m);
/* Код приложения */
}
return 0;
}
Процесс IDLE

Сообщения

- У каждого процесса есть «почтовый ящик» — по умолчанию на одно 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` — блокирующая
  - `msg_try_receive` — неблокирующая
  - `[x,lp,z]timer_msg_receive_timeout` — с максимальным временем ожидания

Mutex

- **Mut**ual **Ex**clusive
- Две операции
  - `mutex_lock` — захват (`mutex_try_lock` — неблокирующая версия); нельзя использовать из прерываний
  - `mutex_unlock` — можно использовать из прерываний и других потоков
- Если поток пытается захватить уже занятый mutex, то он блокируется вплоть до его освобождения
- Инверсия приоритетов — что происходит, если mutex захвачен потоком с низким приоритетом? Как с этим бороться?

The plural of mutex is deadlock

Практика применения

- Mutex'ы используются для управления доступом к периферии (SPI, I^2C и тому подобные интерфейсы)
- Не рекомендуется пользоваться внешними интерфейсами из прерываний
- Обработчик прерывания — максимально простая функция, отправляющая сообщение потоку (или каким-то другим образом разблокирующая его)

Пример: драйвер приемопередатчика NFC ST95HF

- Драйвер и его использование
  - `drivers/st95/st95.[c,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`), вызываются функции `_st95_uart_rx` или `_st95_spi_rx`
- Из обработчиков прерывания передается сообщение потоку

```
static msg_t msg_wu = { .type = UMDK_ST95_MSG_EVENT, };
static void wake_up_cb(void * arg){
    (void) arg;
    msg_try_send(&msg_wu, radio_pid);
}
```

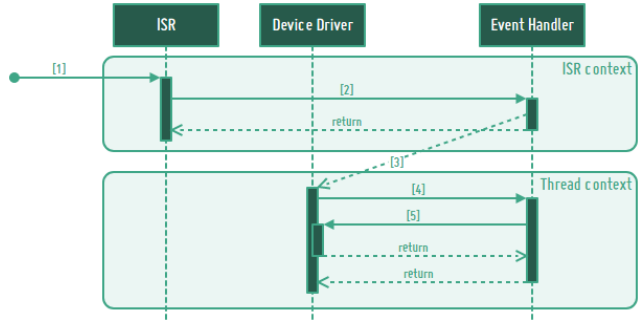
- Уже в этом потоке читаем данные

Пример: интерфейс netdev, драйвер радиомодема SX1276

- SX1276 — радиомодем с поддержкой модуляции LoRa
  - `drivers/sx1276/`
- Реализован стандартный интерфейс netdev

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

Пример: интерфейс netdev



- Создаем поток для обработки прерываний
- Обработчик прерывания (реализован в драйвере) вызывает функцию `dev->event_callback(dev, NETDEV_EVENT_ISR)` (реализует «пользователь»), она разблокирует поток (передает сообщение, освобождает mutex, ...)
- Поток вызывает функцию `isr()` драйвера устройства

Пример: интерфейс netdev, драйвер радиомодема SX1276

```
static void _semtech_loramac_event_cb(netdev_t *dev,
                                      netdev_event_t event) {
    msg_t msg;
    msg.content.ptr = dev;

    switch (event) {
        case NETDEV_EVENT_ISR:
            msg.type = MSG_TYPE_ISR;
            if (msg_send(&msg, semtech_loramac_pid) <= 0) {
                DEBUG("[semtech-loramac] possibly lost interrupt\n");
            }
            break;
            // ...
```

Пример: интерфейс netdev, драйвер радиомодема SX1276

```
void* _semtech_loramac_event_loop(void *arg) {
    msg_init_queue(_semtech_loramac_msg_queue,
                   SEMTECH_LORAMAC_MSG_QUEUE);
    semtech_loramac_t *mac = (semtech_loramac_t *)arg;
    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 {
            switch (msg.type) {
                case MSG_TYPE_RX_TIMEOUT:
                // ...
```