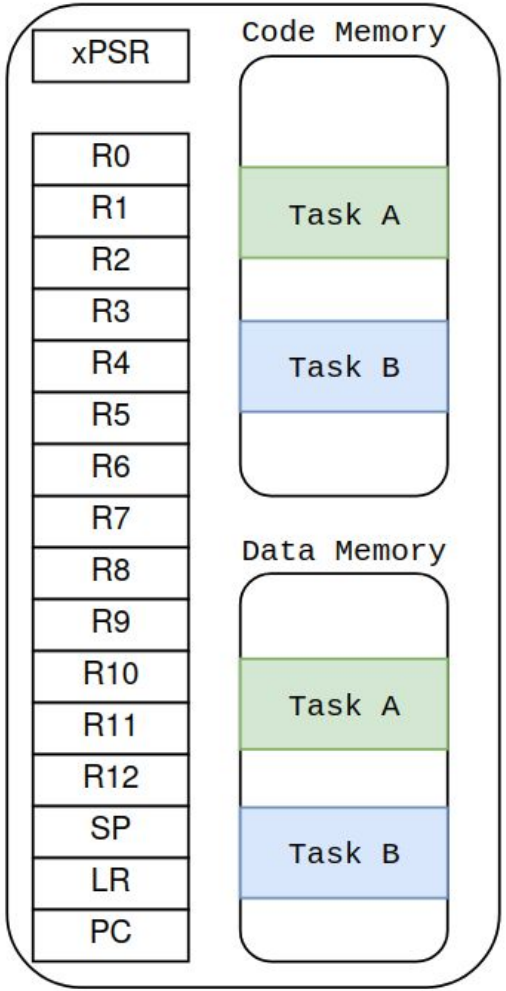
Embedded Operating Systems Design

План:

- Что такое контекст программы/процесса/задачи
- Как имплементировать переключение контекста
- Co-Operative vs Preemptive scheduler/планировщик, tickless scheduler
- RT vs General Purpose scheduler

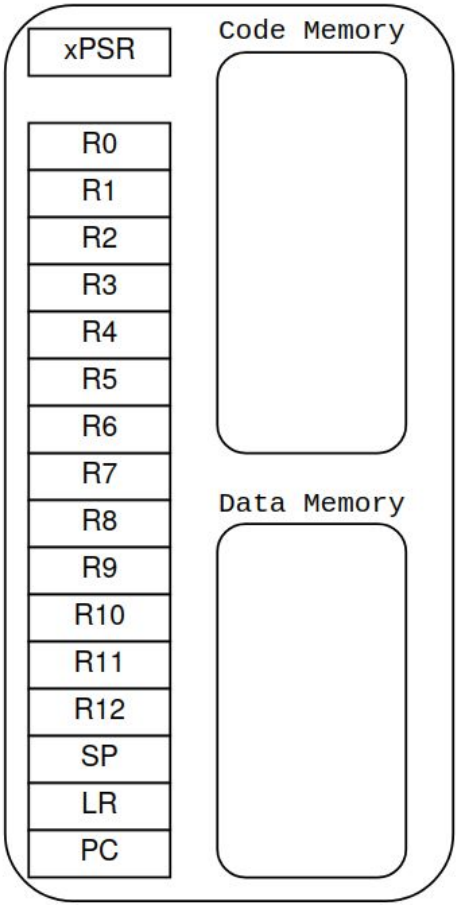
```
void main(void) {
    while (1) {
        taskA();
        taskB();
    }
}
```

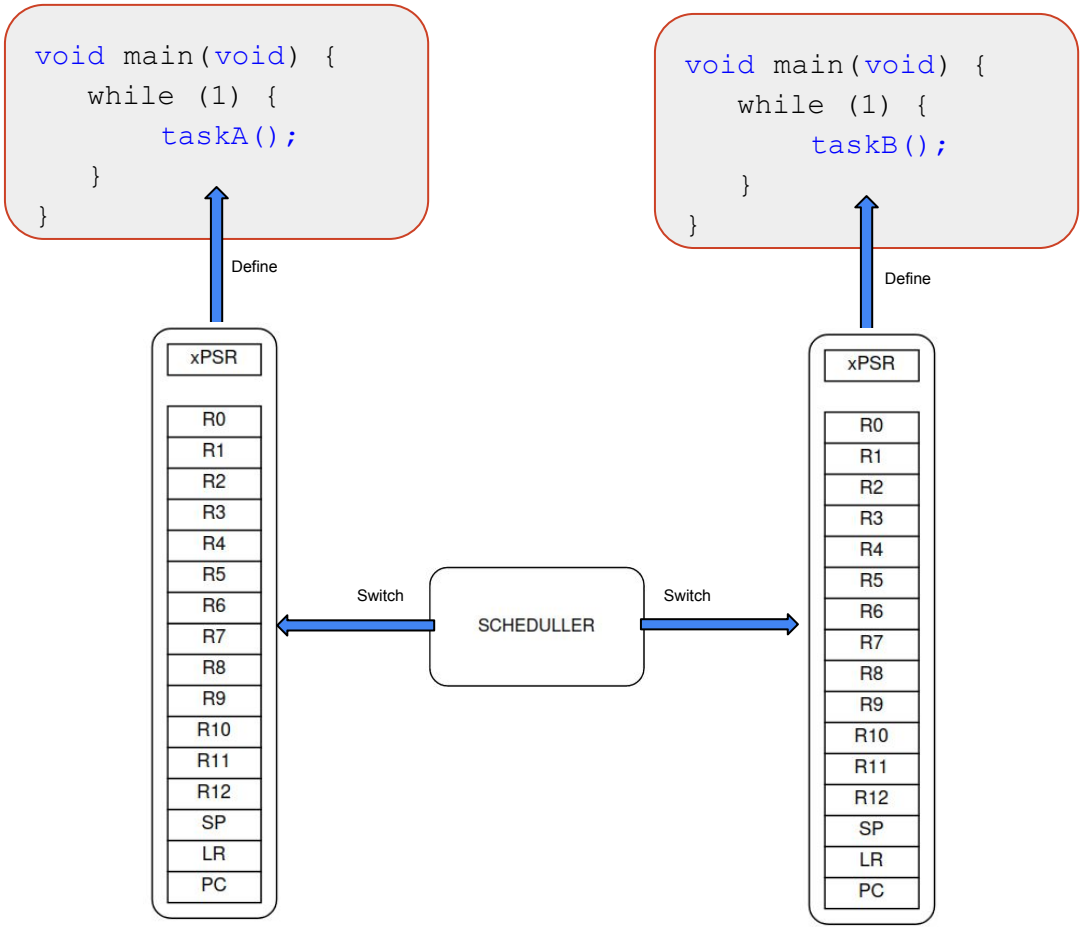


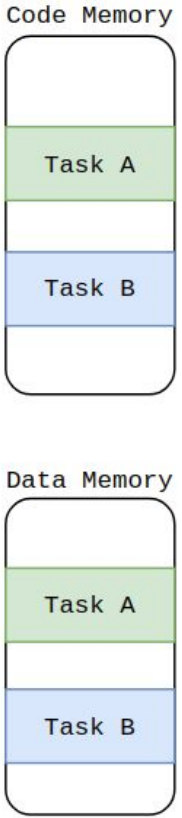

```
void main(void) {
    while (1) {
        taskA();
    }
}
```

```
void main(void) {
    while (1) {
        taskB();
    }
}
```

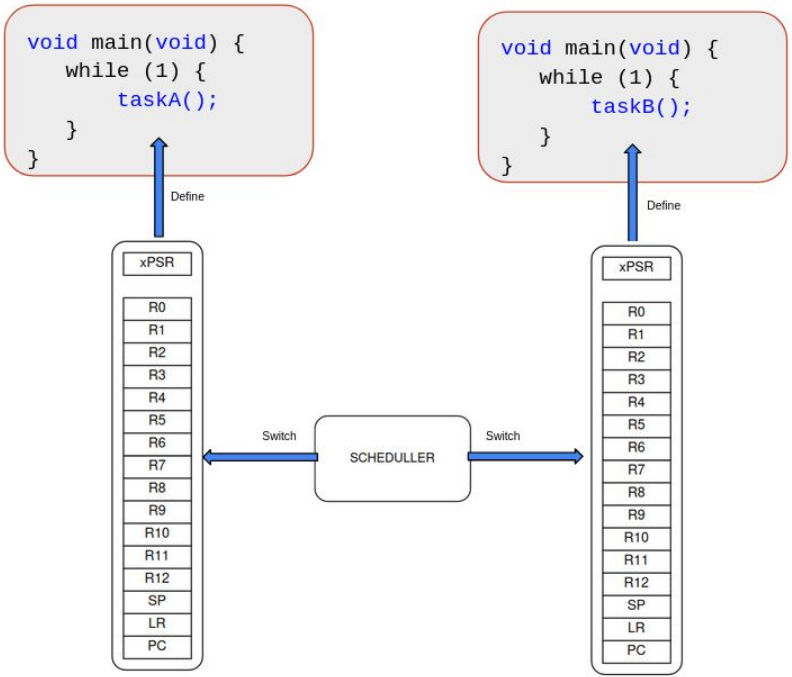
```
int calculate(uint32_t repeat,
              uint32_t x, uint32_t y) {
    int max = 10;
    int res = 0;
    for (int i = 0; i < repeat; ++i) {
        int sum = x + y;
        int mul = x * y;

        res += sum + mul;
        x = (sum < max) ? sum : max;
    }
    return res;
}

void main(void) {
    int res = calculate(10, 2, 4);
}
```

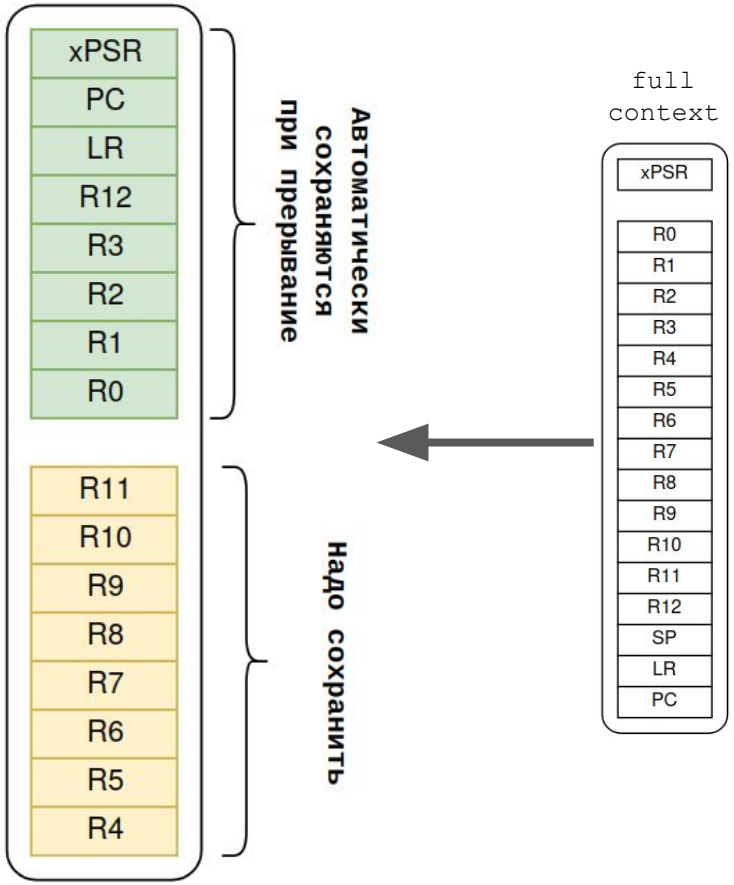


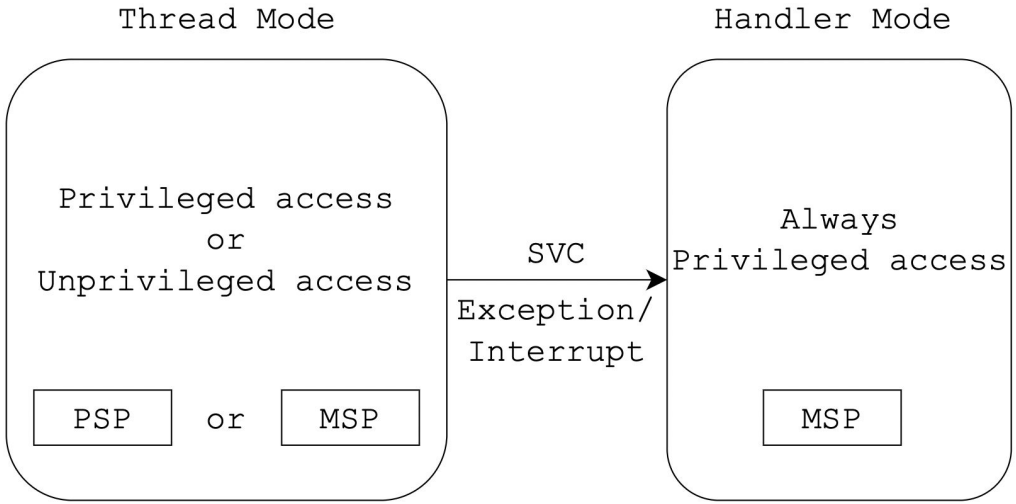


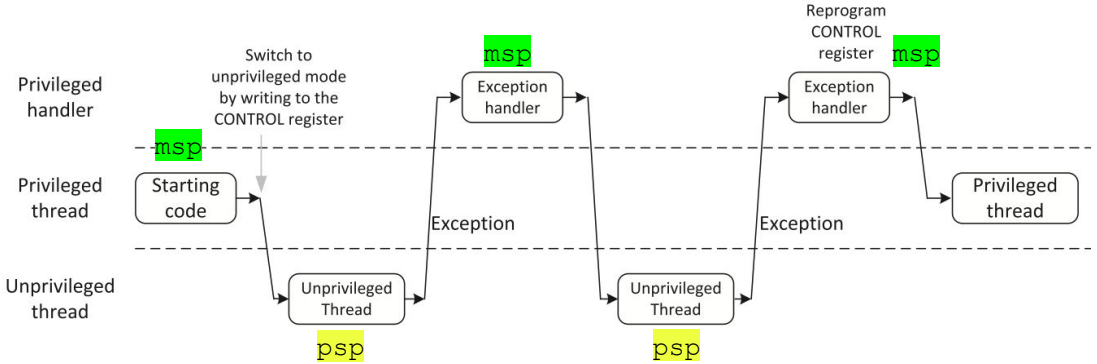


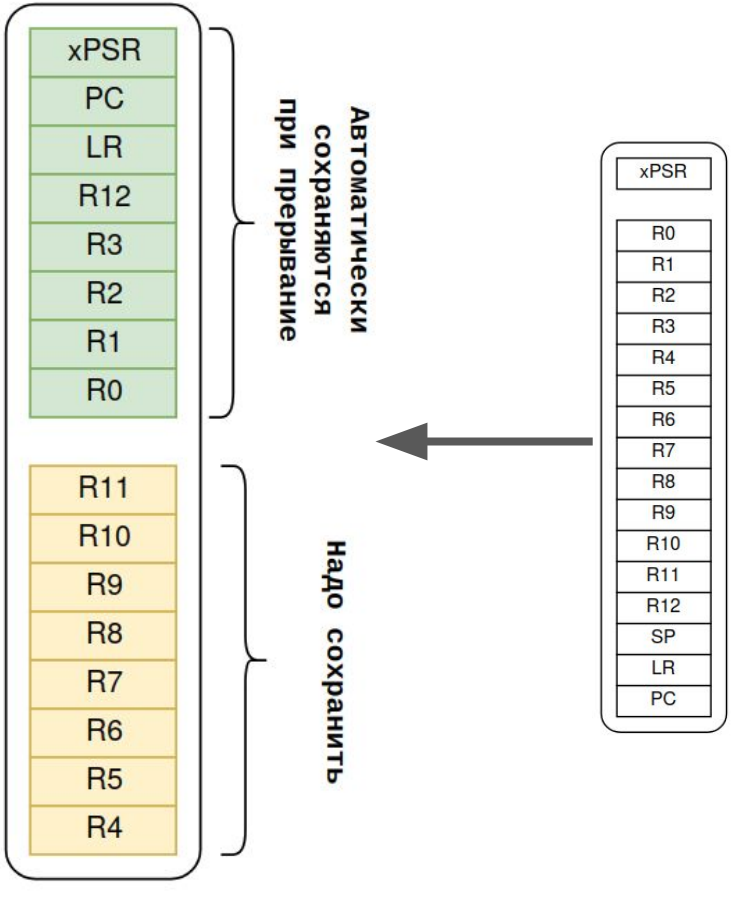
https://www.adamh.cz/blog/2016/07/context-switch-on-the-arm-cortex-m0/


void main(void) {
while (1) {
taskA();
}
}
void main(void) {
while (1) {
taskB();
}
}
Define
Define
xPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
SP
LR
PC
Switch
SCHEDULLER
Switch


xPSR
PC
LR
R12
R3
R2
R1
R0
Автоматически сохраняются при прерывание
R11
R10
R9
R8
R7
R6
R5
R4
Надо сохранить
full context
xPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
SP
LR
PC

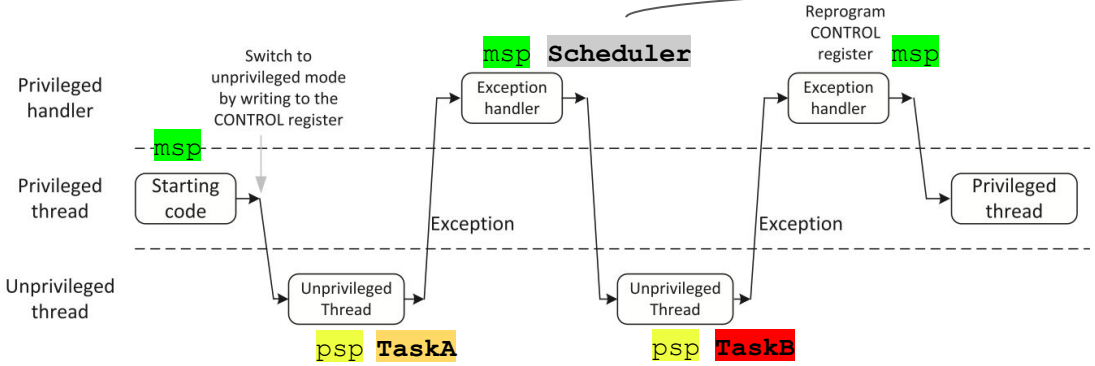

Thread Mode
Handler Mode
Privileged access
or
Unprivileged access
PSP
or
MSP
SVC
Exception/
Interrupt
Always
Privileged access
MSP

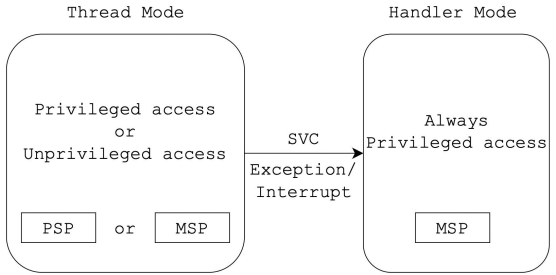

Privileged handler
Privileged thread
Unprivileged thread
msp
Switch to unprivileged mode by writing to the CONTROL register
Starting code
Exception handler
msp
Exception
Unprivileged Thread
psp
Exception handler
Reprogram CONTROL register
msp
Exception
Unprivileged Thread
psp
Privileged thread

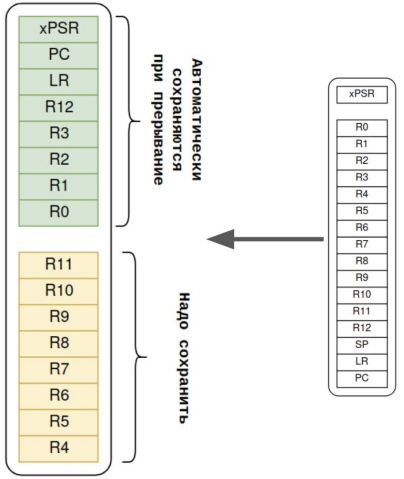

xPSR
PC
LR
R12
R3
R2
R1
R0
Автоматически сохраняются при прерывание
R11
R10
R9
R8
R7
R6
R5
R4
Надо сохранить
xPSR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
SP
LR
PC

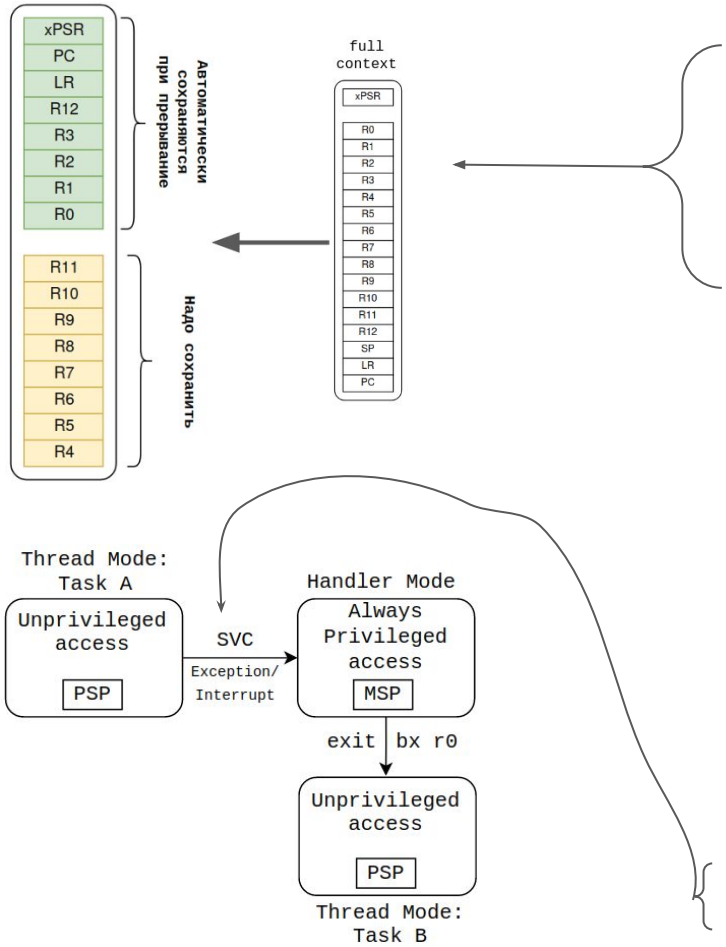
Workflow TaskA -> TaskB Switch:

1. Прерывание прерывает задачу A
2. На прерывание вызывается Scheduler
3. Scheduler сохраняет оставшиеся регистры (R4 - R11) процессора на стэк задачи A
4. Scheduler записывает в процессор новое состояние регистров, соответствующее задаче B
5. Возвращаемся из прерывания в задачу B







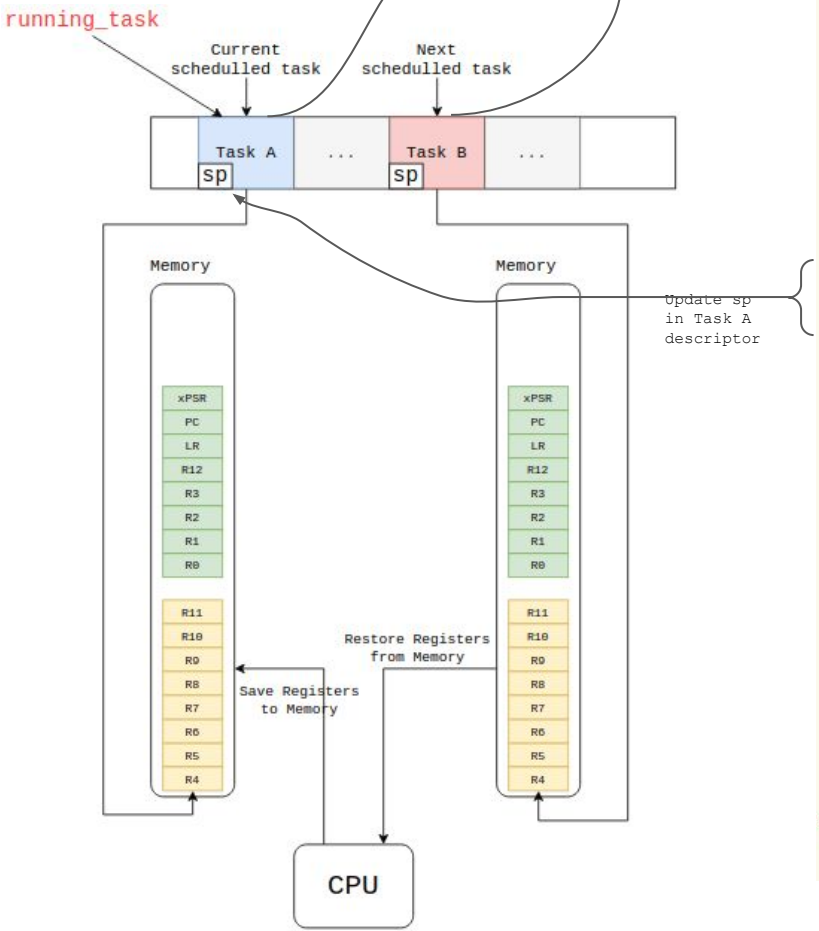



```
SVC_Handler:
  mrs r0, psp   // r0 = psp user mode stack pointer
  sub r0, #32   // allocate 16 byte on the stack for r4-r11 registers (8 registers)

  // save r4-r11 registers at the stack - stmia  r0!, {r4-r11}
  str r4, [r0]
  str r5, [r0, #4]
  str r6, [r0, #8]
  str r7, [r0, #12]
  str r8, [r0, #16]
  str r9, [r0, #20]
  str r10, [r0, #24]
  str r11, [r0, #28]

  ldr r2, =running_task // r2 = addess of cur_task pointer
  ldr r1, [r2]      // r1 = get real task address from cur_task, first value in the task is SP
  str r0, [r1]      // rewrite/save SP address with new in cur_task

  bl ChangeRunningTask // Update cur_task to new/next task

  ldr r2, =running_task
  ldr r1, [r2]
  ldr r0, [r1]      // r0 = SP for cur_task(new/next task)

  // restore r4-r11 registers from the stack - ldmia  r0!,{r4-r11}
  ldr r4, [r0]
  ldr r5, [r0, #4]
  ldr r6, [r0, #8]
  ldr r7, [r0, #12]
  ldr r8, [r0, #16]
  ldr r9, [r0, #20]
  ldr r10, [r0, #24]
  ldr r11, [r0, #28]
  add r0, #32

  msr psp, r0

  ldr r0, =0xFFFFFFFD
  bx  r0

SysCall:
  svc #0x00
  bx lr
```

https://github.com/badembed/cortexM_Simple_Scheduler/blob/master/Src/sched.s

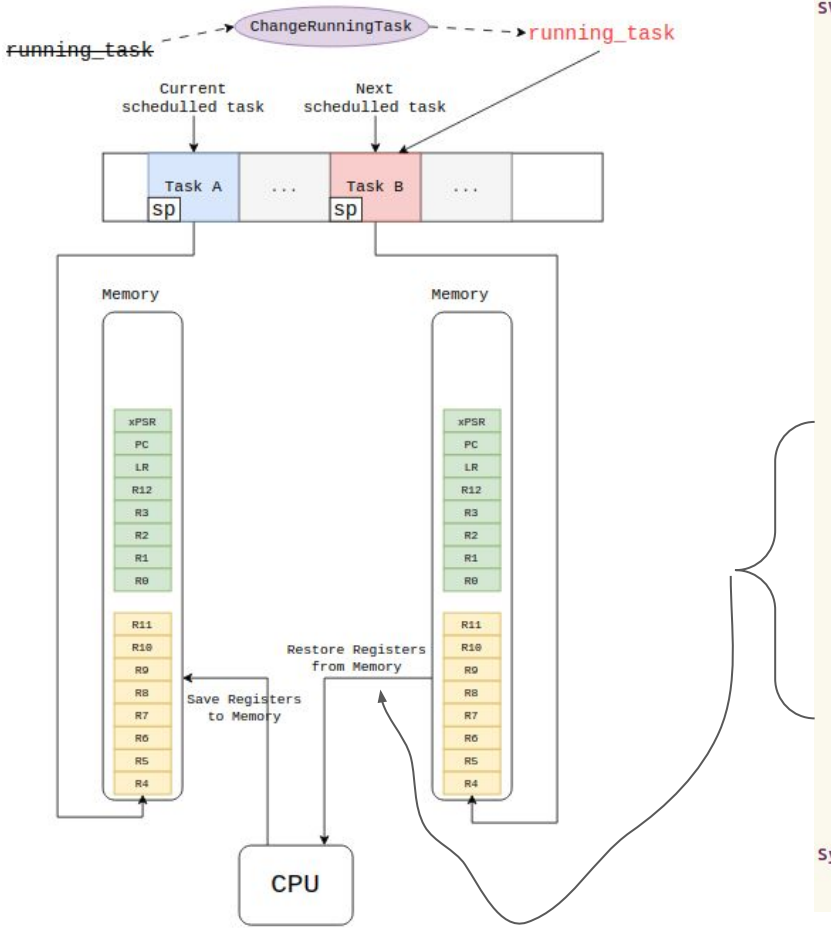
```
volatile tast_t *running_task;

volatile tast_t taskA;
volatile tast_t taskB;
```

```
typedef struct {
  unsigned int sp;
  unsigned int stack[STACK_SIZE];
} tast_t;
```

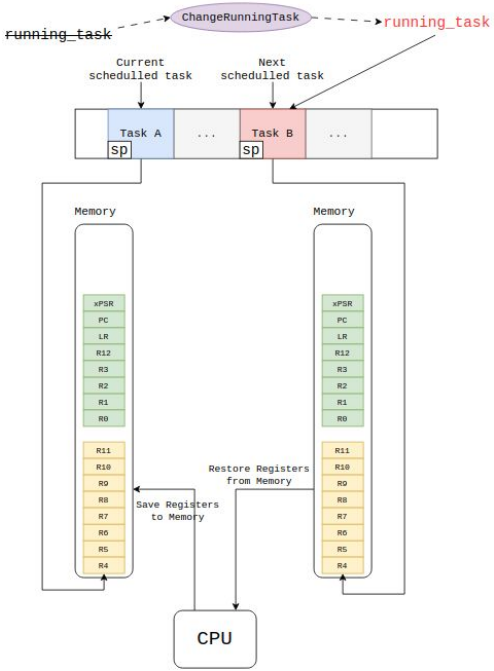


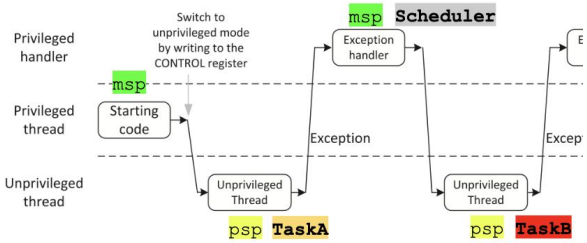

```
SVC_Handler:
  mrs r0, psp   // r0 = psp user mode stack pointer
  sub r0, #32   // allocate 16 byte on the stack for r4-r11 registers (8 registers)

  // save r4-r11 registers at the stack - stmia  r0!, {r4-r11}
  str r4, [r0]
  str r5, [r0, #4]
  str r6, [r0, #8]
  str r7, [r0, #12]
  str r8, [r0, #16]
  str r9, [r0, #20]
  str r10, [r0, #24]
  str r11, [r0, #28]

  ldr r2, =running_task // r2 = addess of cur_task pointer
  ldr r1, [r2]      // r1 = get real task address from cur_task, first value in the task is SP
  str r0, [r1]      // rewrite/save SP address with new in cur_task

  bl ChangeRunningTask // Update cur_task to new/next task

  ldr r2, =running_task
  ldr r1, [r2]
  ldr r0, [r1]      // r0 = SP for cur_task(new/next task)

  // restore r4-r11 registers from the stack - ldmia  r0!,{r4-r11}
  ldr r4, [r0]
  ldr r5, [r0, #4]
  ldr r6, [r0, #8]
  ldr r7, [r0, #12]
  ldr r8, [r0, #16]
  ldr r9, [r0, #20]
  ldr r10, [r0, #24]
  ldr r11, [r0, #28]
  add r0, #32

  msr psp, r0

  ldr r0, =0xFFFFFFFD
  bx  r0

SysCall:
  svc #0x00
  bx lr
```

```
typedef struct {
  unsigned int sp;
  unsigned int stack[STACK_SIZE];
} tast_t;
```




```
SVC_Handler:
  mrs r0, psp   // r0 = psp user mode stack pointer
  sub r0, #32   // allocate 16 byte on the stack for r4-r11 registers (8 registers)

  // save r4-r11 registers at the stack - stmia  r0!, {r4-r11}
  str r4, [r0]
  str r5, [r0, #4]
  str r6, [r0, #8]
  str r7, [r0, #12]
  str r8, [r0, #16]
  str r9, [r0, #20]
  str r10, [r0, #24]
  str r11, [r0, #28]

  ldr r2, =running_task // r2 = addess of cur_task pointer
  ldr r1, [r2]      // r1 = get real task address from cur_task, first value in the task is SP
  str r0, [r1]      // rewrite/save SP address with new in cur_task

  bl ChangeRunningTask // Update cur_task to new/next task

  ldr r2, =running_task
  ldr r1, [r2]
  ldr r0, [r1]      // r0 = SP for cur_task(new/next task)

  // restore r4-r11 registers from the stack - ldmia  r0!,{r4-r11}
  ldr r4, [r0]
  ldr r5, [r0, #4]
  ldr r6, [r0, #8]
  ldr r7, [r0, #12]
  ldr r8, [r0, #16]
  ldr r9, [r0, #20]
  ldr r10, [r0, #24]
  ldr r11, [r0, #28]
  add r0, #32

  msr psp, r0

  ldr r0, =0xFFFFFFFD
  bx  r0

SysCall:
  svc #0x00
  bx lr
```

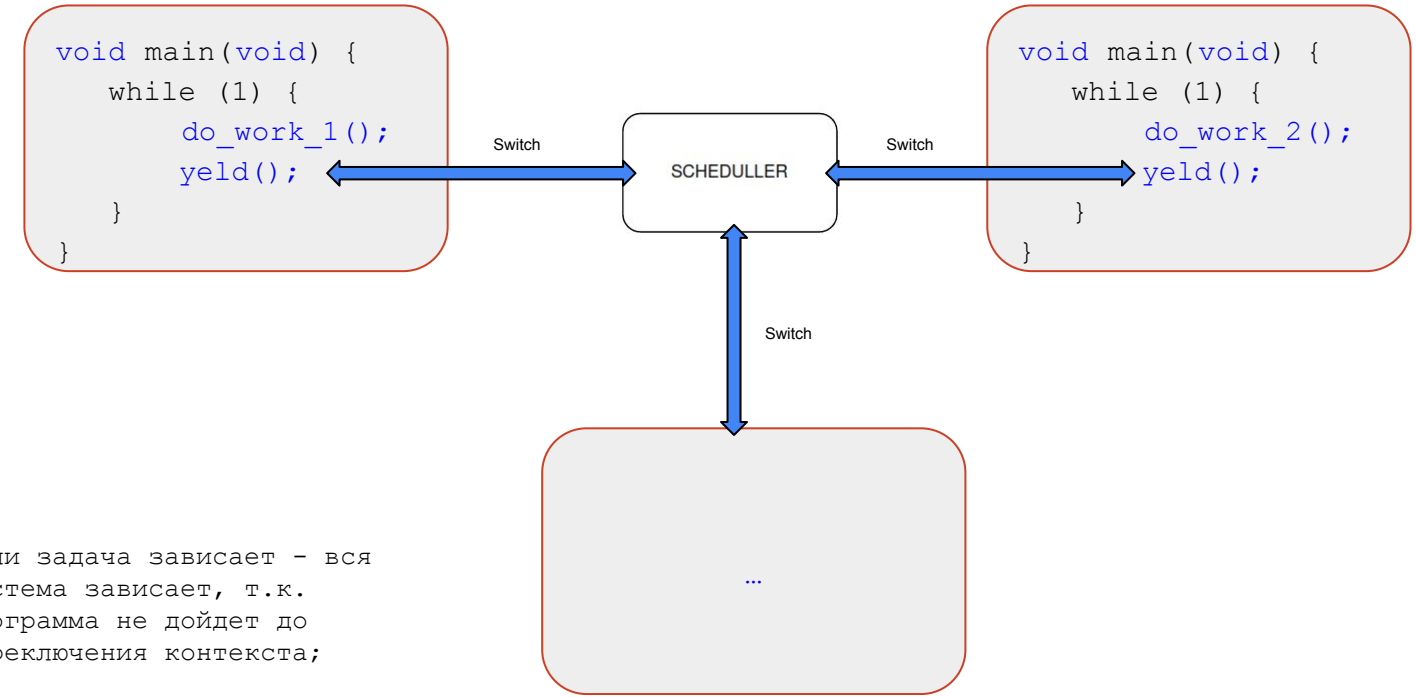




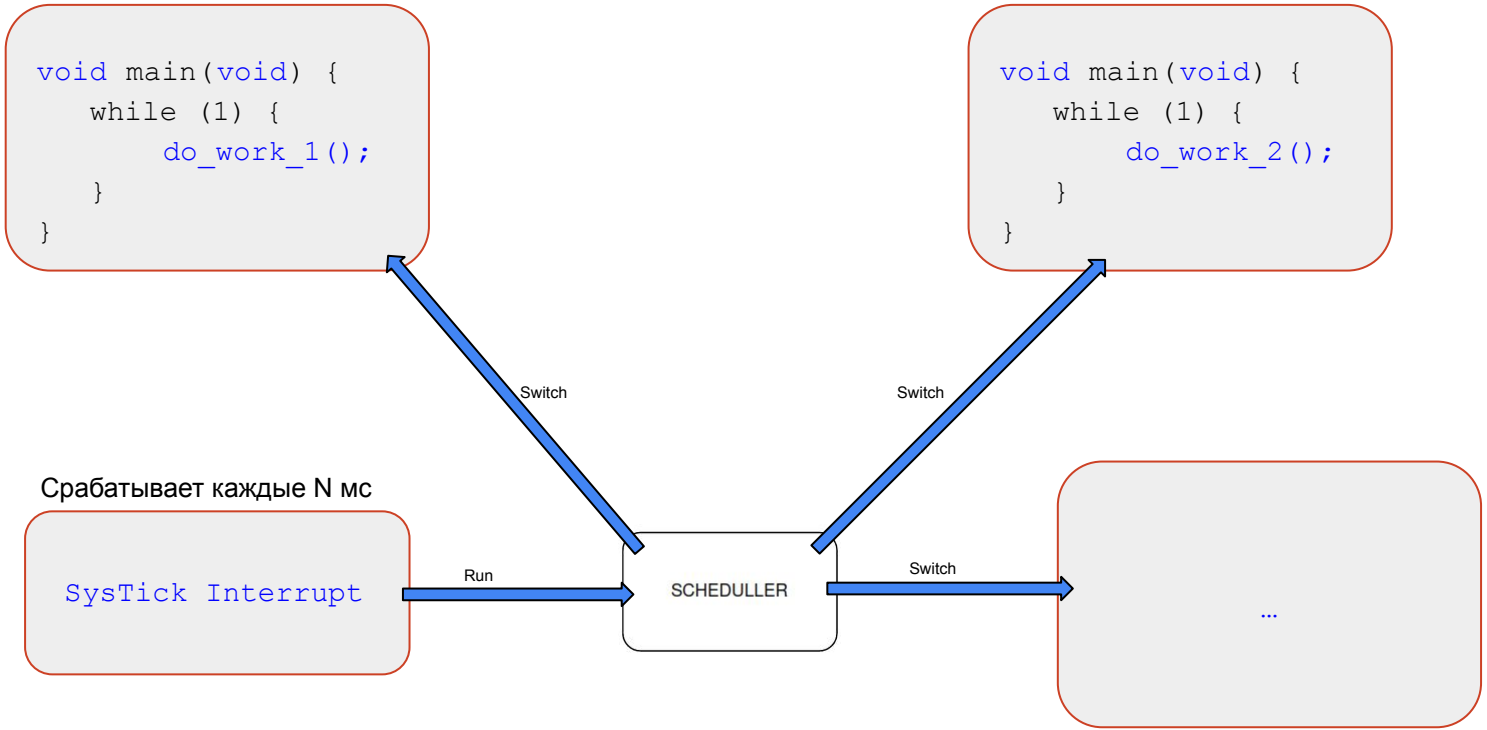

```c
typedef struct {
  unsigned int sp;
  unsigned int stack[STACK_SIZE];
} tast_t;

volatile tast_t *sched_current_task;
volatile tast_t *sched_next_task;
volatile tast_t *running_task;

volatile tast_t taskA;
volatile tast_t taskB;

int cntA;
int cntB;

void TaskA(void)
{
  while(1) {
    cntA++;
    SysCall();
  }
}

void TaskB(void)
{
  while(1) {
    cntB++;
    SysCall();
  }
}

void ChangeRunningTask(void)
{
  running_task = sched_next_task;

  // task schedulling: round-robin for 2 tasks
  tast_t *tmp = sched_current_task;
  sched_current_task = sched_next_task;
  sched_next_task = tmp;
}
```

```c
int main(void)
{
  taskA.stack[STACK_SIZE-1] = 0x01000000UL; // xPSR
  taskB.stack[STACK_SIZE-1] = 0x01000000UL; // xPSR

  taskA.stack[STACK_SIZE-2] = (unsigned int)TaskA | 1;
  taskB.stack[STACK_SIZE-2] = (unsigned int)TaskB | 1;

  // Fill registers with stub values only for debug #
  //taskA.stack[STACK_SIZE-3] = LR; // old saved LR
  taskA.stack[STACK_SIZE-4] = 12; // saved R12
  taskA.stack[STACK_SIZE-5] = 3; // saved R3
  taskA.stack[STACK_SIZE-6] = 2; // saved R2
  taskA.stack[STACK_SIZE-7] = 1; // saved R1
  taskA.stack[STACK_SIZE-8] = 0; // saved R0
  taskA.stack[STACK_SIZE-9] = 11; // saved R11
  taskA.stack[STACK_SIZE-10] = 10; // saved R10
  taskA.stack[STACK_SIZE-11] = 9; // saved R9
  taskA.stack[STACK_SIZE-12] = 8; // saved R8
  taskA.stack[STACK_SIZE-13] = 7; // saved R7
  taskA.stack[STACK_SIZE-14] = 6; // saved R6
  taskA.stack[STACK_SIZE-15] = 5; // saved R5
  taskA.stack[STACK_SIZE-16] = 4; // saved R4

  //taskB.stack[STACK_SIZE-3] = LR; // old saved LR
  taskB.stack[STACK_SIZE-4] = 12; // saved R12
  taskB.stack[STACK_SIZE-5] = 3; // saved R3
  taskB.stack[STACK_SIZE-6] = 2; // saved R2
  taskB.stack[STACK_SIZE-7] = 1; // saved R1
  taskB.stack[STACK_SIZE-8] = 0; // saved R0
  taskB.stack[STACK_SIZE-9] = 11; // saved R11
  taskB.stack[STACK_SIZE-10] = 10; // saved R10
  taskB.stack[STACK_SIZE-11] = 9; // saved R9
  taskB.stack[STACK_SIZE-12] = 8; // saved R8
  taskB.stack[STACK_SIZE-13] = 7; // saved R7
  taskB.stack[STACK_SIZE-14] = 6; // saved R6
  taskB.stack[STACK_SIZE-15] = 5; // saved R5
  taskB.stack[STACK_SIZE-16] = 4; // saved R4
  // ################################################

  taskA.sp = &taskA.stack[STACK_SIZE-16];
  taskB.sp = &taskB.stack[STACK_SIZE-16];

  sched_next_task = &taskB;
  sched_current_task = &taskA;
  running_task = sched_current_task;

  __NVIC_EnableIRQ(SVCall_IRQn);
  /* Switch to unprivileged mode with PSP stack */
  __set_PSP((uint32_t)(&(taskA.stack[STACK_SIZE-16])));
  __set_CONTROL(0x03);

  TaskA();
```

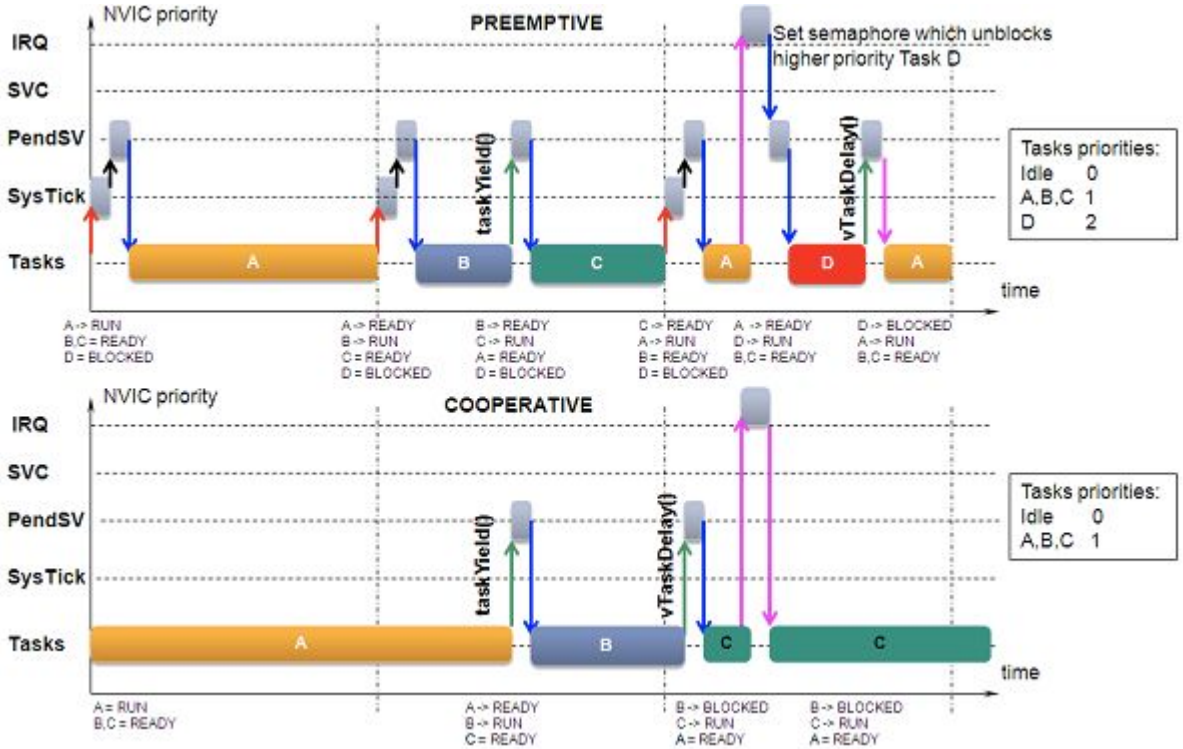
https://github.com/badembed/cortexM_Simple_Scheduler/blob/master/Src/main.c

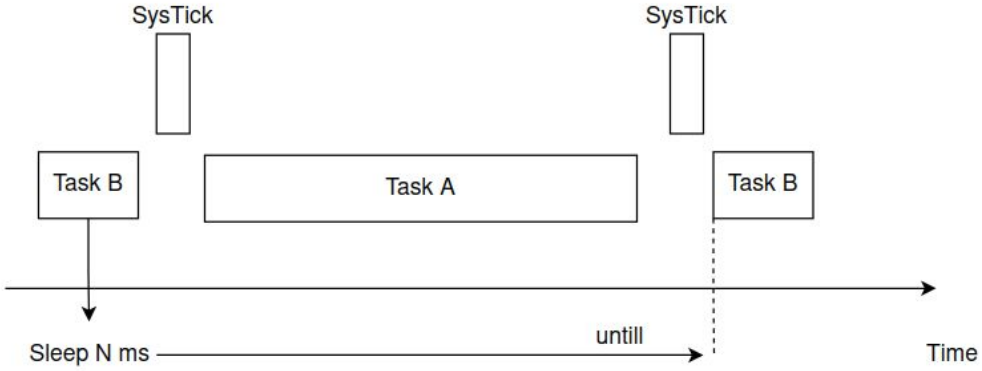


- Если задача зависает - вся система зависает, т.к. программа не дойдет до переключения контекста;



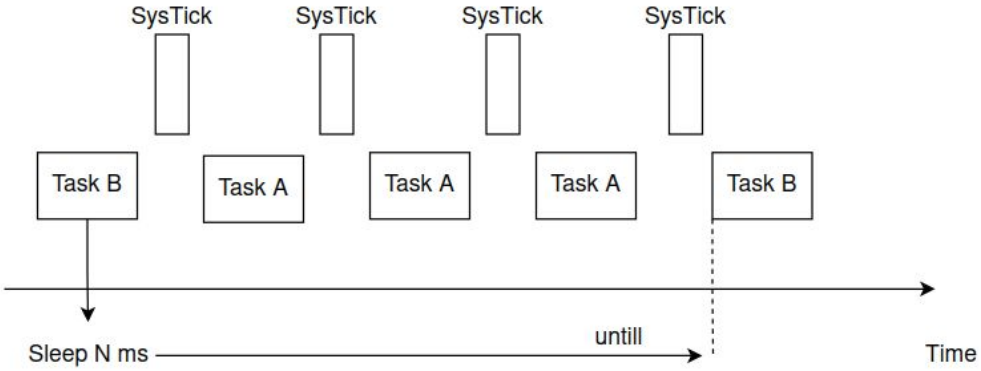
- Если задача зависает - остальные задачи продолжают выполняться, т.к. scheduler будет периодически вызываться по прерыванию таймера и переключать контекст на другие задачи;


NVIC priority
PREEMPTIVE
IRQ
SVC
PendSV
SysTick
Tasks
Set semaphore which unblocks higher priority Task D
taskYield()
vTaskDelay()
A
B
C
A
D
A
time
Tasks priorities:
Idle 0
A,B,C 1
D 2
A -> RUN
B,C = READY
D = BLOCKED
A -> READY
B -> RUN
C = READY
D = BLOCKED
B -> READY
C -> RUN
A = READY
D = BLOCKED
C -> READY
A -> RUN
B = READY
D = BLOCKED
A -> READY
D -> RUN
B,C = READY
D -> BLOCKED
A -> RUN
B,C = READY
NVIC priority
COOPERATIVE
IRQ
SVC
PendSV
SysTick
Tasks
taskYield()
vTaskDelay()
A
B
C
C
time
Tasks priorities:
Idle 0
A,B,C 1
A = RUN
B,C = READY
A -> READY
B -> RUN
C = READY
B -> BLOCKED
C -> RUN
A = READY
B -> BLOCKED
C -> RUN
A = READY



TickLess Scheduling

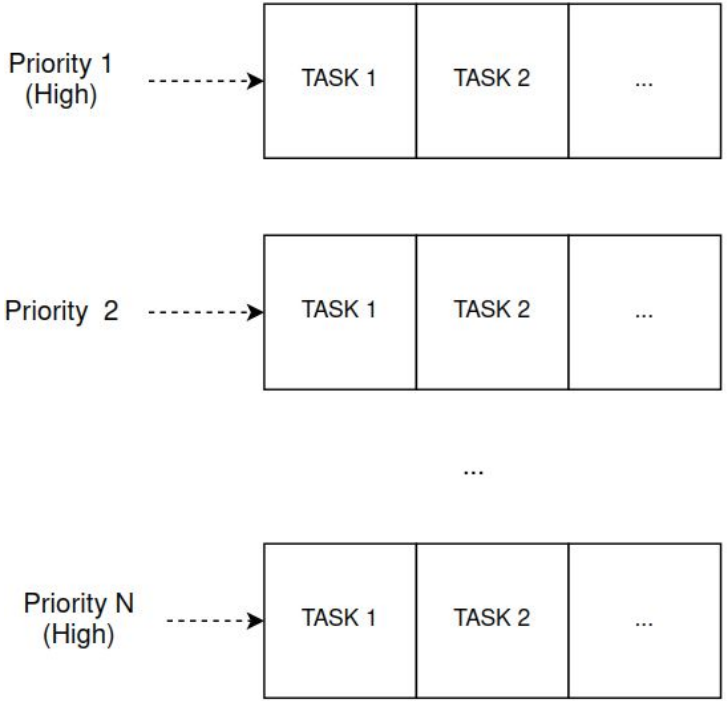
Мы можем заранее вычислять некоторые события и заводить таймер следующего SysTick на большее время.



Example

Если у нас только 2 задачи и мы знаем что **Task B** спит в течении ближайших 2 тиков SysTick, нет необходимости вызывать планировщик т.к. опять **Task A** будет запланирована.

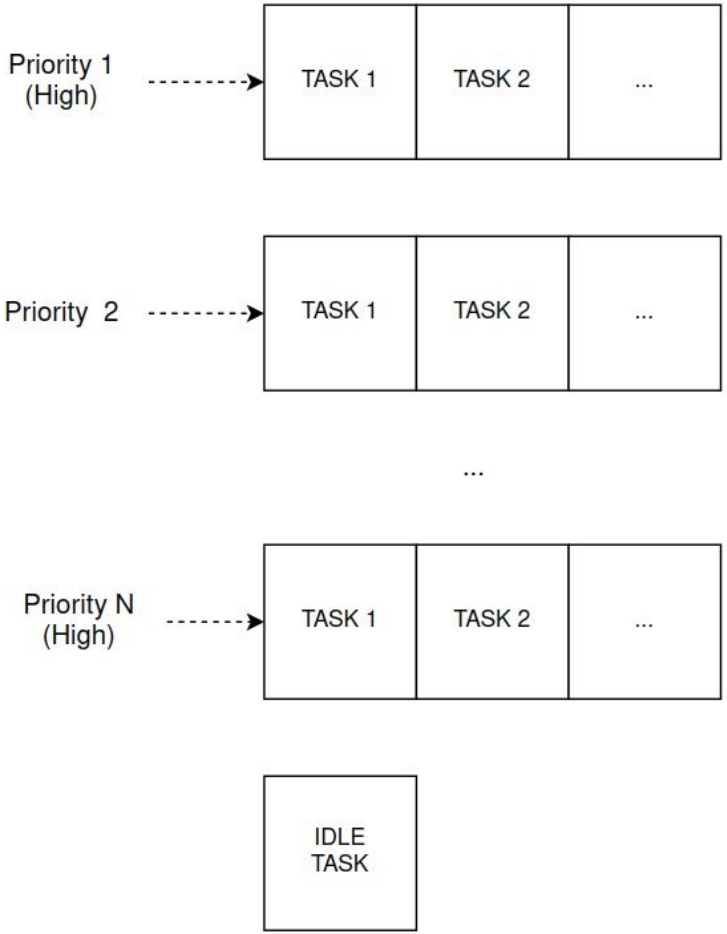
Как часто вызывать планировщик/SysTick ?




```c
typedef struct _TN_TCB
{
   unsigned int * task_stk;    //-- Pointer to task's top of stack
   unsigned int * stk_start;   //-- Base address of task's stack space
   int    stk_size;            //-- Task's stack size (in sizeof(void*),not bytes)
   void * task_func_addr;      //-- filled on creation  (ver 2.x)
   void * task_func_param;     //-- filled on creation  (ver 2.x)
   int  priority;              //-- Task current priority
   int  task_state;            //-- Task state
} TN_TCB;

extern TN_TCB * tn_curr_run_task;       //-- Task that  run now
extern TN_TCB * tn_next_task_to_run;    //-- Task to be run after switch context
```

RTOS Scheduling: Выполняем всегда наиболее приоритетную задачу, вне зависимости от того, как долго она уже выполнялась на процессоре.

IDLE задача выполняется когда нету готовых задач на выполнение. IDLE может переводить процессор в спящий режим (пониженного энергопотребления).



Как планируются задачи с одинаковым приоритетом ?

Например **Round Robin**: Выделяется определенный slice времени - например 100 мс. Первая задача в списке, работает эти 100 мс, а после уходит в конец списка. Затем следующая задача из этого списка планируется и также может отработать 100 мс и после уйти в конец списка и т.д.

General Purpose Scheduling: Дать каждой задаче повыполняться на процессоре. Приоритет задач определяет как быстро задача получит доступ к процессору. **НО,** это не значит что наиболее приоритетная задача всегда выполняется на процессоре.

Нужно учитывать и время которое задача уже выполнялась на процессоре и ее приоритет.

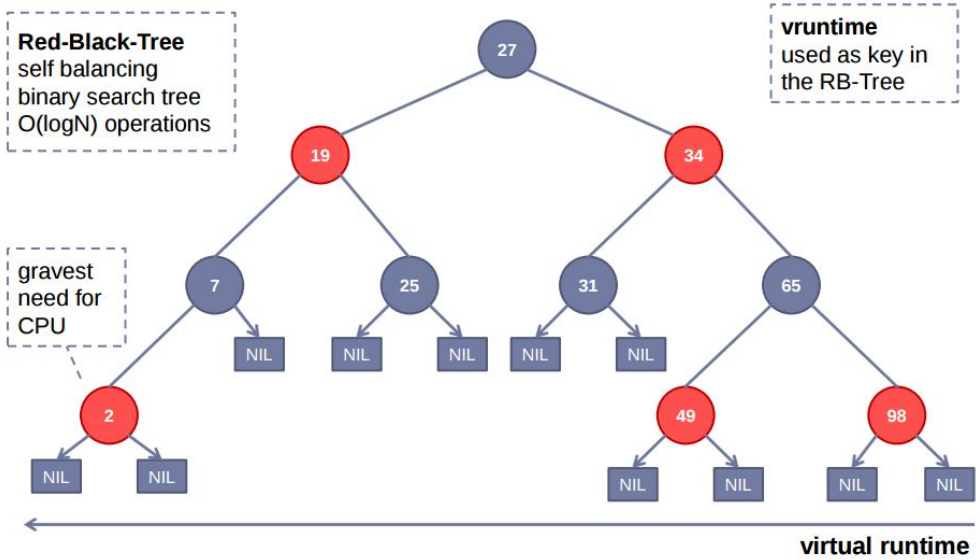
IO Bound Task: Text Editor	**CPU Bound Task**: Video Encoder
Тратит почти все время на ожидание события - нажатия кнопки от пользователя. В это время задача в режиме wait и не выполняется на процессоре.	Тратит все время на обсчет на данных на процессоре.
Короткое время выполнения - не требует много процессорного времени на выполнение самой задачи.	Выполняется достаточно долго - требует много процессорного времени на выполнение самой задачи.
При возникновении события, нужно как можно раньше дать доступ к процессору, чтобы пользователь не ждал.	Не критично к времени запуска задачи.
При возникновении события, нужно дать достаточно процессорного времени чтобы закончить задачу сразу.	Не критично к времени окончания задачи. Может быть вытеснена Text Editor’ом чтобы дать ему обработать его событие.

General Purpose Scheduling: Дать каждой задаче повыполняться на процессоре.
Запускать задачу, которая, на данный момент, выполнялась наименьшее время на процессоре.
Нужно учитывать и время которое задача уже выполнялась на процессоре и ее приоритет.

```
vruntime = time * nice_value
time - сколько задача уже выполнялась на процессоре
nice_value - коэффициент зависящий от приоритета задачи. Задачи с высоким приоритетом имеют более низкие значения
nice_value, что приводит к тому что vruntime нарастает медленнее.
```

Планировщик выбирает на выполнение задачу с наименьшим показателем **vruntime.**

Таким образом Text Editor, который выполняется меньшее время на процессоре - всегда будет вытеснять Video Encoder, который выполняется на процессоре долго.



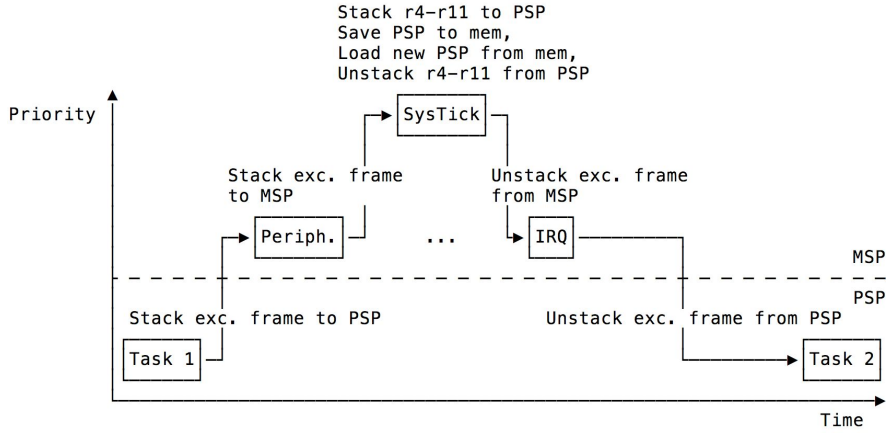
https://www.cse.iitd.ac.in/~rijurekha/col788/scheduling1.pdf
https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

Что вынести из этой лекции:

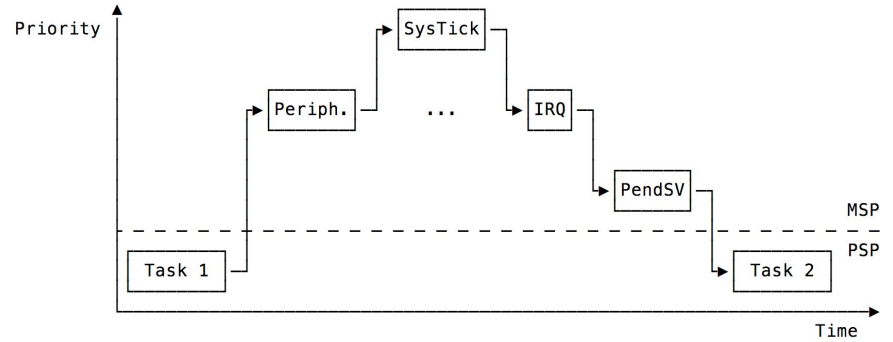
- Что такое контекст задачи/процесса
- Переключение контекста
- Планировщики

Задание:

- Взять scheduler https://github.com/badembed/cortexM_Simple_Scheduler и добавить в него 3ю задачу и приоритеты задач. Сделать так чтобы ChangeRunningTask планировала наиболее приоритетную задачу. Сделать логику понижения приоритетов (например чтобы когда задача отработала какое то время у нее понижался приоритет), чтобы в вашей тестовой программе все 3 задачи поработали.



Если SysTick прерывание произошло во время обработки другого прерывания: при первом прерывании R3-R0 сохранятся на стэк PSP, и в процессе выполнения обработчика прерывания R4-R11 могут измениться - а в это время придет SysTick прерывание и сохранит новые (некорректные для задачи) R4-R11 регистры на PSP stack…



Для решения данной проблемы в Cortex-M есть PendSV прерывание, которое имеет минимальный приоритет. На него заводится смена контекста (SysTick выставляет PendSV внутри), и тогда смена контекста произойдет только после обработки всех прерываний !!!

https://www.adamh.cz/blog/2016/07/context-switch-on-the-arm-cortex-m0/