

**Федеральное государственное автономное образовательное учреждение
высшего образования**

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Московский институт электроники и математики им. Тихонова

Департамент электронной инженерии

ОТЧЕТ

О ПРАКТИЧЕСКОЙ РАБОТЕ №7

по дисциплине «Цифровая схемотехника и архитектура компьютера»

«Средства диспетчеризации»

Студенты гр. БИБ201

Кречетов Андрей

Шадрунов Алексей

Дата выполнения: 19 марта 2023 г.

Преподаватель:

к. т. н., доцент кафедры

информационной безопасности

киберфизических систем

Мещеряков Я. Е.

«___» _____ 2023 г.

Москва, 2023

Содержание

1	Цель работы	3
2	Теоретические сведения	4
2.1	Классификация диспетчеров по стратегии обслуживания	5
2.2	Классификация по способу распределения процессорного времени . . .	5
2.3	Стратегии диспетчеризации	6
2.3.1	Стратегия «в порядке живой очереди»	6
2.3.2	Стратегия «первый с кратчайшими сроками выполнения работы / следующий с кратчайшим заданием»	6
2.3.3	Стратегия «следующий по наименьшему остающемуся времени»	7
2.3.4	Стратегия «следующий с наибольшим относительным временем ответа»	7
2.3.5	Стратегия «Round Robin»	8
2.3.6	Стратегия диспетчеризации по приоритету	8
3	Ход работы	9
3.1	Компиляция	9
3.2	Организация вывода	9
3.3	Организация ввода	10
3.4	Результат работы программы	10
3.5	Дополнительные пункты	12
4	Выводы о проделанной работе	15
	Список использованных источников	16
	Приложение А. Код вытесняющего диспетчера	17

1 Цель работы

Изучить механизмы создания и управления процессами в кооперативном и вытесняющим режиме; изучить внутреннее устройство организации кооперативной и вытесняющей диспетчеризаций. С помощью примитивных средств диспетчеризации реализовать решение задачи из варианта для микроконтроллеров семейства AVR.

2 Теоретические сведения

Диспетчеризация процессора — предоставление всем процессам в системе по очереди в определенном порядке квантов процессорного времени.

Главной целью диспетчеризации является **максимальная и равномерная** загрузка процессора.

Работа любого процесса в системе представляется как последовательность чередований фаз активности процессора и активности ввода-вывода. Частота периодов активности процессора обратно пропорциональна их длительности.

Операционная система — это комплекс программ, предназначенных для управления ресурсами компьютера и организации взаимодействия с пользователем. Операционная система выполняет следующие основные функции, связанные с управлением задачами:

- создание и удаление задач;
- планирование процессов и диспетчеризация задач;
- синхронизация задач, обеспечение их средствами коммуникации.

Планировщик — компонента ОС, планирующая выделение квантов времени процессам по определенной стратегии. Различаются стратегии с прерыванием процессов (**вытесняющие**) (когда при вводе нового более короткого или более приоритетного процесса в систему текущий процесс прерывается) и без прерывания процессов (**кооперативные**).

Диспетчер — компонента ОС, выполняющая переключение процессора с одного процесса на другой. Время, которое на это требуется, называется скрытой активностью (латентностью) диспетчера и должно быть минимизировано.

Диспетчеризация — задача динамического (краткосрочного) планирования процессов для наиболее эффективного распределения ресурсов, возникающих практически при каждом событии.

Основные критерии диспетчеризации:

1. Максимизация использования процессора;
2. Максимизация пропускной способности системы;
3. Минимизация среднего времени обработки одного процесса;
4. Минимизация среднего времени ожидания одним процессом;
5. Минимизация среднего времени ответа системы;
6. Равномерность загрузки процессора;

2.1 Классификация диспетчеров по стратегии обслуживания

- Беспriorитетное обслуживание. Выбор задачи производится в некотором заранее установленном порядке без учета их относительной важности и времени обслуживания.

- Приоритетное обслуживание. При обслуживании отдельных задач, задачам предоставляется преимущественное право попасть в состояние исполнения.

Приоритет, присвоенный задаче, может быть постоянным или изменяться в процессе её решения.

2.2 Классификация по способу распределения процессорного времени

- Диспетчеризация без перераспределения процессорного времени, то есть невытесняющая многозадачность (non-preemptive multitasking, кооперативная диспетчеризация) — способ диспетчеризации процессов, при котором активный процесс выполняется до своего завершения. Дисциплины обслуживания FCFS, SJN, SRT относятся к невытесняющим.

- Диспетчеризация с перераспределением процессорного времени (вытесняющая многозадачность, preemptive multitasking) — способ, при котором решение о переключении процессора принимается диспетчером задач. Механизм диспетчеризации задач целиком сосредоточен в операционной системе, и программист может писать свое приложение, не заботясь о том, как оно будет выполняться параллельно с другими задачами. При этом операционная система выполняет следующие функции:

- определяет момент снятия с выполнения текущей задачи, сохраняет её контекст в дескрипторе задачи (или в отведенном сегменте памяти)

- выбирает из очереди готовых задач следующую и запускает её на выполнение, предварительно загрузив её контекст

Дисциплина RR и многие другие, построенные на её основе, относятся к вытесняющим. Число переключений контекста с процесса на процесс возрастает с уменьшением выделяемого кванта времени. Для обработки процессов различных классов и приоритетов (например, пакетных и интерактивных) ОС создает многоуровневые аналитические очереди процессов, каждая из которых обслуживается по различным стратегиям и предоставляет процессам кванты времени различного размера. Процесс может быть переведен из одной очереди в другую.

2.3 Стратегии диспетчеризации

2.3.1 Стратегия «в порядке живой очереди»

First Come First Served (FCFS) — предоставление ресурсов процессам в порядке их ввода в систему независимо от длительности. Время ожидания может оказаться большим, особенно если первым в систему вводится более длительный процесс (эффект сопровождения).

Образуются две очереди: одна из новых задач, а вторая — из ранее выполнявшихся, но попавших в состояние ожидания (рисунок 1). Это позволяет по возможности заканчивать вычисления в порядке их появления. Эта дисциплина обслуживания не требует внешнего вмешательства в ход вычислений, при ней не происходит перераспределение процессорного времени.

Достоинства: простота реализации и малые расходы ресурсов на формирование очереди задач.

Недостатки: при увеличении загрузки вычислительной системы растет и среднее время ожидания обслуживания, причем короткие задания вынуждены ожидать столько же, сколько и трудоёмкие. Избежать этого недостатка позволяют дисциплины SJN и SRT.

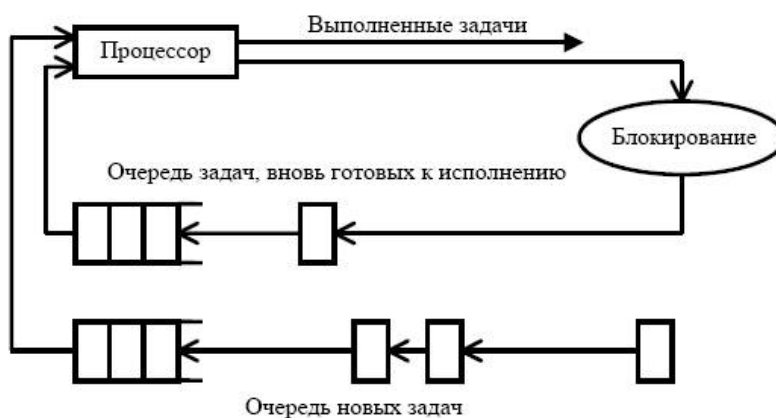


Рисунок 1 – Стратегия First Come First Served

2.3.2 Стратегия «первый с кратчайшими сроками выполнения работы / следующий с кратчайшим заданием»

Shortest Job First (SJF) / Shortest Job Next (SJN) — дисциплина планирования без переключения, согласно которой следующим для выполнения выбирается ожидающий процесс с наименьшим временем выполнения — SRTF (Shortest-Remaining-Time-First). Данная стратегия обеспечивает минимальное среднее время ожидания процессов и является кооперативной стратегией.

2.3.3 Стратегия «следующий по наименьшему остающемуся времени»

Shortest remaining time next / Shortest remaining time first, SRTN/SRTF — вытесняющая версия стратегии SJN, где процессор выделяется для задачи, у которой время ближе всего к завершению (оставшееся время). Применяется в системах с разделением времени.

Метод экспоненциального усреднения позволяет вычислить предсказываемую длину следующего периода активности по фактическим и предсказанным длинам предыдущих периодов активности. Оставшееся время — это разность между временем, запрошенным пользователем (временем выполнения), и временем, которое процесс уже получил и которое измеряется системой с помощью аппаратного таймера. По принципу SRTN/SRTF первым всегда выполняется процесс, имеющий минимальное оценочное время до завершения, причем с учетом новых поступающих процессов. Если в соответствии с алгоритмом SPN процесс, запущенный в работу, выполняется до своего завершения, то по алгоритму SRT процесс может быть прерван при поступлении нового процесса, имеющего более короткое оценочное время работы. Чтобы механизм SRT был эффективным, нужны достаточно точные оценки будущего.

Механизм SRT характеризуется более высокими накладными расходами по сравнению с SPN, так как должен следить за текущим временем обслуживания выполняющегося задания и обрабатывать возникающие прерывания. Поступающие в систему небольшие процессы будут выполняться почти немедленно. Однако более длительные процессы будут иметь даже большее среднее время ожидания и большую дисперсию времени ожидания, чем в случае SPN.

Реализация принципа SRT требует, чтобы регистрировались истекшие временные интервалы обслуживания задач, а это приводит к увеличению накладных расходов. Теоретически алгоритм SRT обеспечивает минимальные времена ожидания. Однако из-за издержек на переключения может оказаться так, что в определенных ситуациях в действительности лучшие показатели будет иметь SPN.

2.3.4 Стратегия «следующий с наибольшим относительным временем ответа»

Highest Response Ratio Next, HRRN — стратегия диспетчеризации без переключения, согласно которой приоритет каждого процесса является не только функцией времени выполнения этого процесса, но также времени, затраченного процессом на ожидание выполнения. После того, как процесс получает в свое распоряжение процес-

сор, он выполняется до завершения. Приоритеты при дисциплине HRRN вычисляются по формуле:

Приоритет = (время ожидания + время выполнения) / время выполнения.

Поскольку время выполнения находится в знаменателе, предпочтение будет оказываться более коротким процессам. Однако, поскольку в числителе имеется время ожидания, более длинные процессы, которые уже довольно долго ждут, будут также получать определенное предпочтение.

Алгоритм компенсирует некоторые из недостатков SPN, например, чрезмерную задержку в обслуживании длинных процессов и чрезмерно быстрый отклик на новые короткие задачи.

2.3.5 Стратегия «Round Robin»

Стратегия Round Robin предоставляет всем процессам по очереди одинаковые кванты времени процессора (рисунок 2). Квант времени не должен быть слишком мал, иначе накладные расходы на переключение процессов оказываются сравнимыми с полезным временем процессора. Стратегия RR обеспечивает лучшее время ответа, чем SJF, но худшее время оборота.

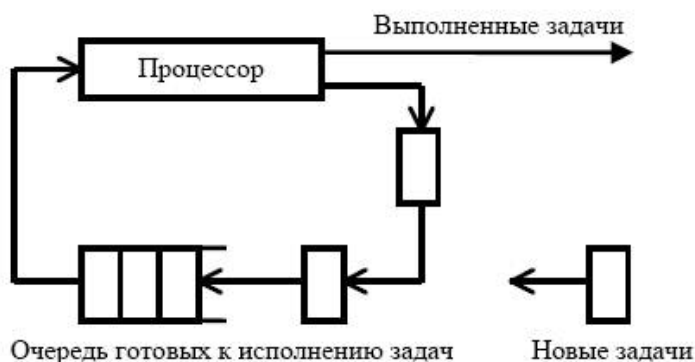


Рисунок 2 – Стратегия Round Robin

2.3.6 Стратегия диспетчеризации по приоритету

Стратегия диспетчеризации по приоритету предоставляет первым ресурсы процессора более высокоприоритетному процессу. Чтобы избежать ситуации "голодания", ОС постепенно повышает приоритеты процессов, длительное время находящихся в системе.

3 Ход работы

3.1 Компиляция

Рассмотрим реализацию программы. Разработка и отладка полностью реализованы на ОС Linux под архитектуру atmega88 с помощью инструмента с открытым исходным кодом `simavr`, библиотеки `libxc` (см. Список использованных источников) и предоставленной реализации `SimpleDispatcher` [1] [2].

Команды для компиляции решения представлены в листинге 1.

```
avr-gcc -I/home/sthussky/libxc/include/ -Wall -mmcu=atmega88 -c -x c
    -funsigned-char -funsigned-bitfields -Og -ffunction-sections
    -fdata-sections -fpack-struct -fshort-enums -g2 -Wall -MD -MP -MF -c
    main.c -o main.o
avr-gcc -I/home/sthussky/libxc/include/ -Wall -mmcu=atmega88 -c -x c
    -funsigned-char -funsigned-bitfields -Og -ffunction-sections
    -fdata-sections -fpack-struct -fshort-enums -g2 -Wall -MD -MP -MF -c
    rtos.c -o rtos.o
avr-gcc -I/home/sthussky/libxc/include/ -Wall -mmcu=atmega88 -c -x c
    -funsigned-char -funsigned-bitfields -Og -ffunction-sections
    -fdata-sections -fpack-struct -fshort-enums -g2 -Wall -MD -MP -MF -c
    software_RTC.c -o srtc.o -fcommon
avr-gcc -o dispatch88.elf rtos.o srtc.o main.o -lm -mmcu=atmega88
    -funsigned-char -funsigned-bitfields -Wl,--start-group -Wl,-lm
    -Wl,--end-group -Wl,--gc-sections -Og -ffunction-sections
    -fdata-sections -fpack-struct -fshort-enums
    -Wl,-Map="simpleDispatcher.map" -fcommon
```

Листинг 1 – Команды для компиляции

3.2 Организация вывода

Вывод программы реализован как симуляция взаимодействия с UART-интерфейсом. Это позволяет в реальном времени отслеживать состояние программы, не пользуясь средствами отладки. Настройка взаимодействия с интерфейсом приведена на рисунке 3.

```

static int uart_putchar(char c, FILE *stream)
{
    if (c == '\n')
        uart_putchar('\r', stream);
    loop_until_bit_is_set(UCSR0A, UDRE0);
    UDR0 = c;
    return 0;
}

static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar, NULL,
                                          _FDEV_SETUP_WRITE);

volatile uint8_t bindex = 0;
uint8_t buffer[80];
volatile uint8_t done = 0;

ISR(USART_RX_vect)
{
    uint8_t b = UDR0;
    GPIOR1 = b; // for the trace file
    buffer[bindex++] = b;
    buffer[bindex] = 0;
    if (b == '\n')
        done++;
}

```

Рисунок 3 – Настройка взаимодействия с симуляционным UART-интерфейсом

3.3 Организация ввода

Согласно требованиям к лабораторной работе, файл с входными данными реализован как массив во flash-памяти микроконтроллера (листинг 2). Чтение из него реализовано функцией `pgm_read_byte()`. Подобные инициализация и чтение возможны благодаря библиотеке `avr/pgmspace.h`

```

const char data[255] PROGMEM =
    "tW#B5^hknk941D*!8NwH6DOVhs%od(Aa6D7[LOq4t@(jdd<.VeY7#N1W6l&5[l1QFQbRw5#Nv1@
    Fd6k8Qz143@00W$p5W9%/4W8[{ydhTXNF<{1e#Qx3y9%/st.9)lIxUje4AjH=w!t1+p79AWSk3}{
    W6Bb_3[@>tzqff3$c2vL/m_WXt-I4yj-&Qgd)*nbstP1R$9e63G>u#3tJU99r0/q3548^jjR3u1
    %t]D*zr66X^47MsK2c8.O1[";

```

Листинг 2 – Массив входных данных

3.4 Результат работы программы

Результатом работы программы является полностью работоспособная программа на основе кооперативного диспетчера. При корректно указанных в тексте программы входных данных она выдает верный результат (рисунок 4). Диаграмма Ганта приведена на рисунке 5, блок-схема — на рисунке 6.

```
sthussky@xor ~/d/C/simpleDispatcher> simavr -m atmega88 dispatch88.elf
Loaded 3418 bytes at 0
Loaded 100 bytes at 800100
Starting count 100 - 250..
Finishing count 100 - 250 with result 6..
Starting count 0 - 100..
Finishing count 0 - 100 with result 2..
Overall result: 8..
sthussky@xor ~/d/C/simpleDispatcher> 
```

Рисунок 4 – Корректная работа программы

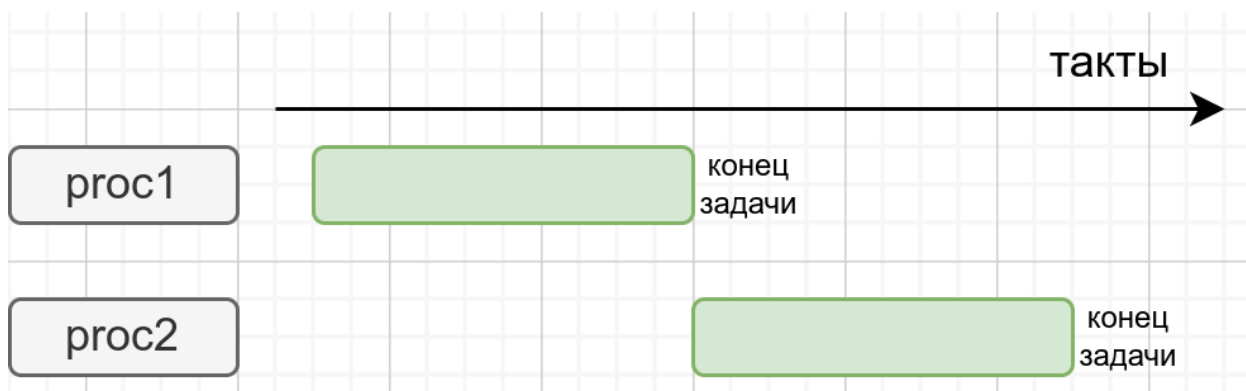


Рисунок 5 – Диаграмма

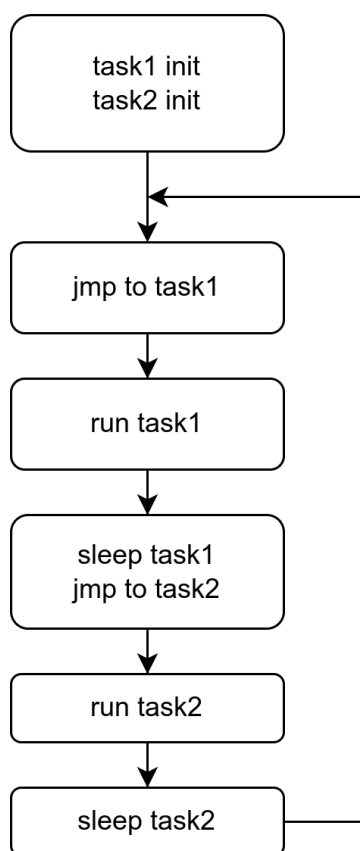
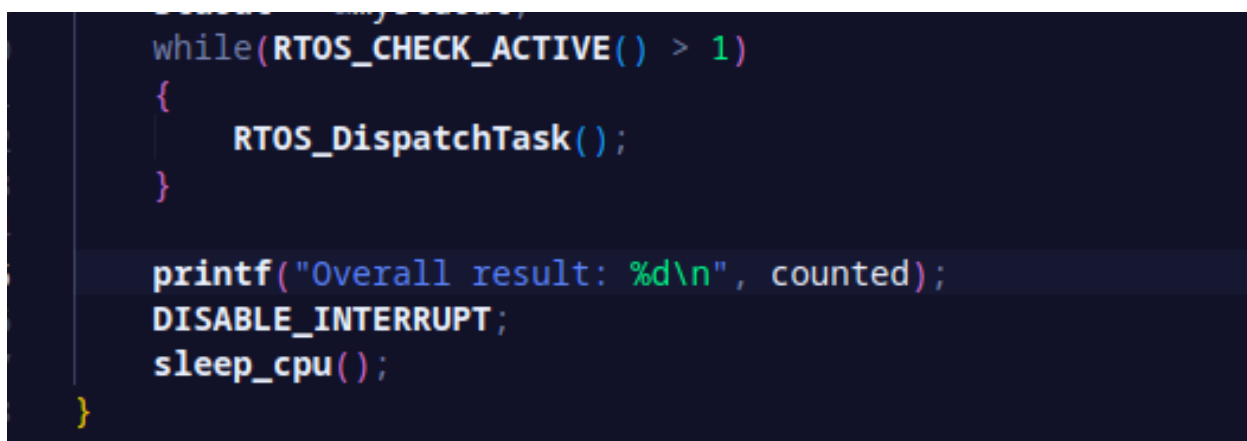


Рисунок 6 – Блок-схема

3.5 Дополнительные пункты

После изучения средства симуляции `simavr` реализовано завершение программы с помощью двух команд — `DISABLE_INTERRUPT` и `sleep_cpu` (библиотека `avr/sleep.h`). Здесь используется особенность `simavr` — симуляция прекращается при вечном (никак не прерываемом) `sleep_mode` процессора.

Для получения критерия завершения в модуле `RTOS` реализован счетчик задач — `volatile static uint8_t` переменная, инкрементируемая при вызове `RTOS_SetTask` и декрементируемая при вызове `RTOS_DeleteTask`. Также реализован интерфейс для нее — функция `RTOS_CHECK_ACTIVE()`, возвращающая значение 0 при условии наличия лишь одной активной задачи (взведенного таймера). Логика в `main.c` принимает вид:



```
while(RTOS_CHECK_ACTIVE() > 1)
{
    RTOS_DispatchTask();
}

printf("Overall result: %d\n", counted);
DISABLE_INTERRUPT;
sleep_cpu();
}
```

Рисунок 7 – Завершение программы

В качестве дополнительного задания с помощью открытых источников реализовано простое подобие вытесняющего диспетчера без использования `DisplacingDispatcher` [3]. Результат работы реализации представлен на рисунке 8:

```
Loaded 462 bytes at 800100
Starting..
0 from proc1 read 0 - 5..
1 from proc2 read 50 - 55..
0 from proc1 read 5 - 10..
0 from proc2 read 55 - 60..
0 from proc1 read 10 - 15..
0 from proc2 read 60 - 65..
0 from proc1 read 15 - 20..
0 from proc2 read 65 - 70..
0 from proc1 read 20 - 25..
0 from proc2 read 70 - 75..
0 from proc1 read 25 - 30..
0 from proc2 read 75 - 80..
0 from proc1 read 30 - 35..
1 from proc2 read 80 - 85..
0 from proc1 read 35 - 40..
0 from proc2 read 85 - 90..
0 from proc1 read 40 - 45..
1 from proc2 read 90 - 95..
1 from proc1 read 45 - 50..
Goodbye from proc1, subtotal 4..
1 from proc2 read 95 - 100..
Goodbye from proc2, subtotal 5..
```

Рисунок 8 – Независимая реализация вытесняющей логики

Диаграмма Ганта приведена на рисунке 9.

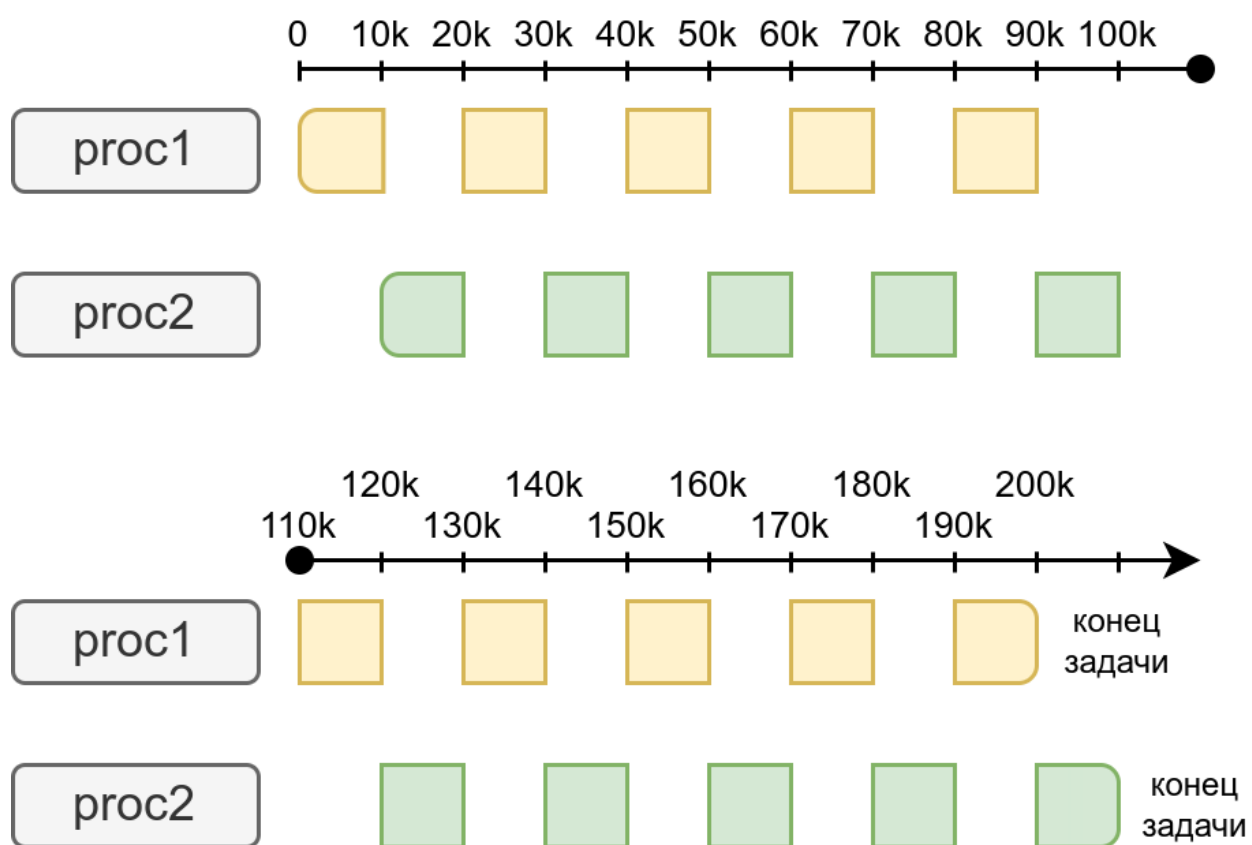


Рисунок 9 – Диаграмма вытесняющего диспетчера

4 Выводы о проделанной работе

В рамках данной работы изучены механизмы создания и управления процессами в кооперативном и вытесняющем режиме; изучено внутреннее устройство организации кооперативной и вытесняющей диспетчеризаций. С помощью примитивных средств диспетчеризации реализовано решение задачи из варианта для микроконтроллеров семейства AVR. Выполнен ряд дополнительных заданий, связанных с улучшением UX при разработке под AVR. Изучены opensource инструменты, полезные при такой разработке.

Список использованных источников

- [1] Susi Lehtola Miguel A. L. Marques, Micael Oliveira. Libxc installation guide. <https://www.tddft.org/programs/libxc/installation/>.
- [2] Michel Pollet. simavr — a lean and mean atmel avr simulator for linux. <https://github.com/buserror/simavr>.
- [3] Susi Lehtola Miguel A. L. Marques, Micael Oliveira. qsnake libxc opensource package. <https://github.com/qsnake/libxc>.

Приложение А. Код вытесняющего диспетчера

```
1 /*
2
3     Copyright 2008-2013 Michel Pollet <busererror@gmail.com>
4
5
6     simavr is free software: you can redistribute it and/or modify
7     it under the terms of the GNU General Public License as published by
8     the Free Software Foundation, either version 3 of the License, or
9     (at your option) any later version.
10
11     simavr is distributed in the hope that it will be useful,
12     but WITHOUT ANY WARRANTY; without even the implied warranty of
13     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
14     GNU General Public License for more details.
15
16     You should have received a copy of the GNU General Public License
17     along with simavr. If not, see <http://www.gnu.org/licenses/>.
18 */
19
20 #ifndef __AVR_CR_H__
21 #define __AVR_CR_H__
22
23 /*
24  * Smallest coroutine implementation for AVR. Takes
25  * 23 + (24 * tasks) bytes of SRAM to run.
26  *
27  * Use it like:
28  *
29  * AVR_TASK(mytask1, 32);
30  * AVR_TASK(mytask2, 48);
31  * ...
32  * void my_task_function() {
33  *     do {
34  *         AVR_YIELD(mytask1, 1);
35  *     } while (1);
36  * }
37  * ...
38  * main() {
39  *     AVR_TASK_START(mytask1, my_task_function);
40  *     AVR_TASK_START(mytask2, my_other_task_function);
41  *     do {
42  *         AVR_RESUME(mytask1);
43  *         AVR_RESUME(mytask2);
44  *     } while (1);
45  * }
46  * NOTE: Do not use "static" on the function prototype, otherwise it
47  * will fail to link (compiler doesn't realize the "jmp" is referencing)
48 */
49 #include <setjmp.h>
50 #include <stdint.h>
51 static inline void _set_stack(register void *stack)
52 {
53     asm volatile(
54         "in r0, __SREG__\n"
55         "\n\t"
56         "cli\n"
57         "\n\t"
58         "out __SP_H__, %B0\n"
59         "\n\t"
60         "out __SREG__, r0\n"
61         "\n\t"
62         "out __SP_L__, %A0\n"
63         "\n\t"
64         :
65         : "e"(stack) /* : */
66     );
67 }
```

```

68 jmp_buf g_caller;
69 #define AVR_TASK(_name, _stack_size) \
70     struct \
71     { \
72         jmp_buf jmp; \
73         uint8_t running : 1; \
74         uint8_t stack[_stack_size]; \
75     } _name
76 #define AVR_TASK_START(_name, _entry) \
77     if (!setjmp(g_caller)) \
78     { \
79         _set_stack(_name.stack + sizeof(_name.stack)); \
80         asm volatile("rjmp " #_entry); \
81     }
82 #define AVR_YIELD(_name, _sleep) \
83     _name.running = !_sleep; \
84     if (!setjmp(_name.jmp)) \
85         longjmp(g_caller, 1)
86 #define AVR_RESUME(_name) \
87     if (!setjmp(g_caller)) \
88         longjmp(_name.jmp, 1)
89 #endif /* __AVR_CR_H__ */
90
91 #include <avr/io.h>
92 #include <avr/interrupt.h>
93 #include <avr/sleep.h>
94 #include <stdio.h>
95
96 static int uart_putchar(char c, FILE *stream)
97 {
98     if (c == '\n')
99         uart_putchar('\r', stream);
100     loop_until_bit_is_set(UCSR0A, UDRE0);
101     UDR0 = c;
102     return 0;
103 }
104
105 static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar, NULL,
106                                         _FDEV_SETUP_WRITE);
107
108 char input_file[250] =
109     "R+fF?s?o}aXw72(0)!JO@2119mgEbw*Kv=4cMc&&CD2tEX3t?L#_348#qFn@RO)e8@>z}#KF7f9W[E
109 uint8_t count(char *buf, uint8_t start, uint8_t end, char to_find)
110 {
111     uint8_t count = 0;
112     for (uint8_t j = start; j < end; j++)
113     {
114         if (buf[j] == to_find)
115         {
116             count++;
117         }
118     }
119     return count;
120 }
121
122 AVR_TASK(mytask1, 32);
123 AVR_TASK(mytask2, 32);
124 AVR_TASK(mytask3, 32);
125
126 uint8_t task_stat_1 = 0;
127 uint8_t task_stat_2 = 0;
128
129 uint8_t task_res_1 = 0;
130 uint8_t task_res_2 = 0;
131
132 uint16_t t = 0;
133
134 void smth()
135 {

```

```

136     uint16_t c = 0;
137     do
138     {
139         c++;
140         if (c == 20000)
141         {
142             printf("running\n");
143         }
144         AVR_YIELD(mytask3, 1);
145     } while (1);
146 }
147
148 void count_function1()
149 {
150     uint16_t c = 0;
151     uint8_t start = 0;
152     uint8_t end = 50;
153     uint8_t step = 5;
154     static uint8_t i = 0;
155     do
156     {
157         c++;
158         if (c == 10000)
159         {
160             c = 0;
161             i++;
162             uint8_t curc = count(input_file, start + step * (i - 1), start
+ step * i, '3');
163             task_res_1 += curc;
164             printf("%d from proc1 read %d - %d\n", curc, start + step * (i
- 1), start + step * i);
165
166             if (start + step * i == end)
167             {
168                 printf("Goodbye from proc1, subtotal %d\n", task_res_1 +
task_res_2);
169                 task_stat_1 = (uint8_t)1;
170             }
171         }
172
173         AVR_YIELD(mytask1, 1);
174     } while (task_stat_1 == 0);
175 }
176
177 void count_function2()
178 {
179     uint16_t c = 0;
180     uint8_t start = 50;
181     uint8_t end = 100;
182     uint8_t step = 5;
183     static uint8_t i = 0;
184
185     do
186     {
187         c++;
188         if (c == 10000)
189         {
190             c = 0;
191             i++;
192             uint8_t curc = count(input_file, start + step * (i - 1), start
+ step * i, '3');
193             task_res_2 += curc;
194             printf("%d from proc2 read %d - %d\n", curc, start + step * (i
- 1), start + step * i);
195             if (start + step * i == end)
196             {
197                 task_stat_2 = (uint8_t)1;
198                 printf("Goodbye from proc2, subtotal %d\n", task_res_1 +
task_res_2);

```

```

199         }
200     }
201     AVR_YIELD(mytask2, 1);
202 } while (task_stat_2 == 0);
203 }
204
205 void dispatch()
206 {
207     AVR_TASK_START(mytask1, count_function1);
208     AVR_TASK_START(mytask2, count_function2);
209     AVR_TASK_START(mytask3, smth);
210     static uint8_t total = 0;
211     do
212     {
213         if (task_stat_1 == 0)
214         {
215             AVR_RESUME(mytask1);
216         }
217         if (task_stat_2 == 0)
218         {
219             AVR_RESUME(mytask2);
220         }
221         AVR_RESUME(mytask3);
222     } while (1);
223 }
224
225 int main()
226 {
227     stdout = &mystdout;
228     sei(); // Enable global interrupts
229     printf("Starting\n");
230     dispatch();
231 }

```