The friendly Operating System for IoT

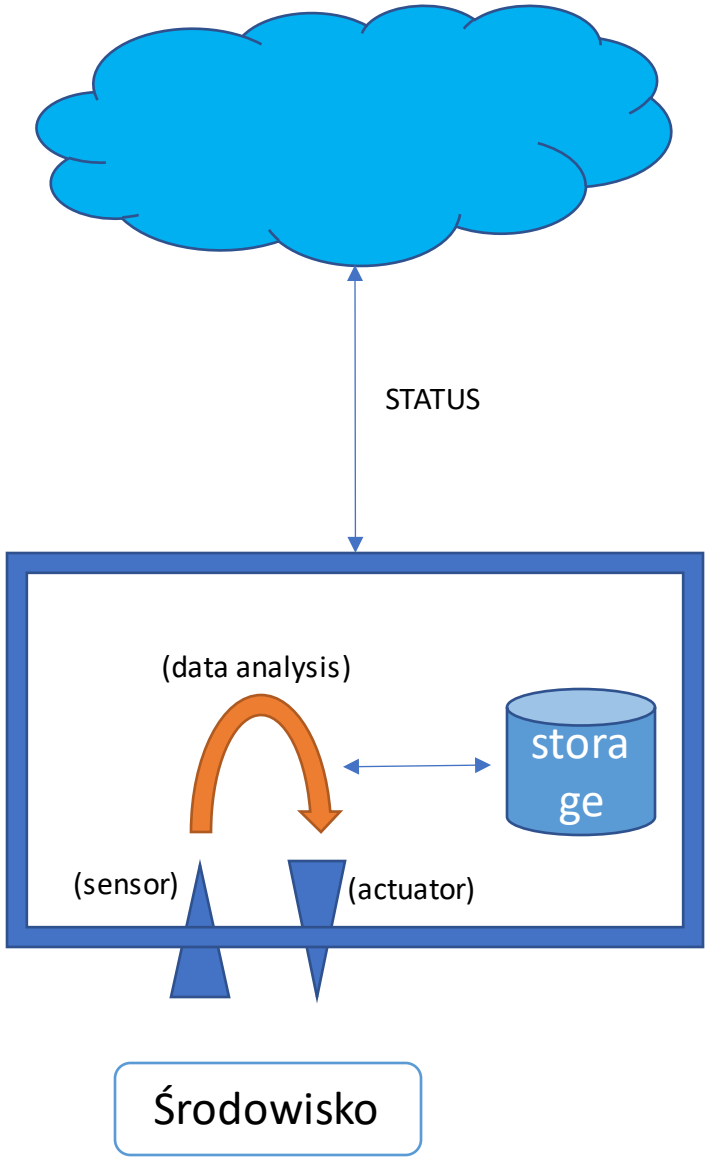
Real Internet of Things Operating System (RIOT-OS)

Krzysztof Pachowicz– k.pachowicz@tele.pw.edu.pl

2022/05/17, PSIR 2021Z


TEMPERATURE
MOISTURE
PRESSURE
HEALTH
CONNECTIVITY

Pattern – Device Computing



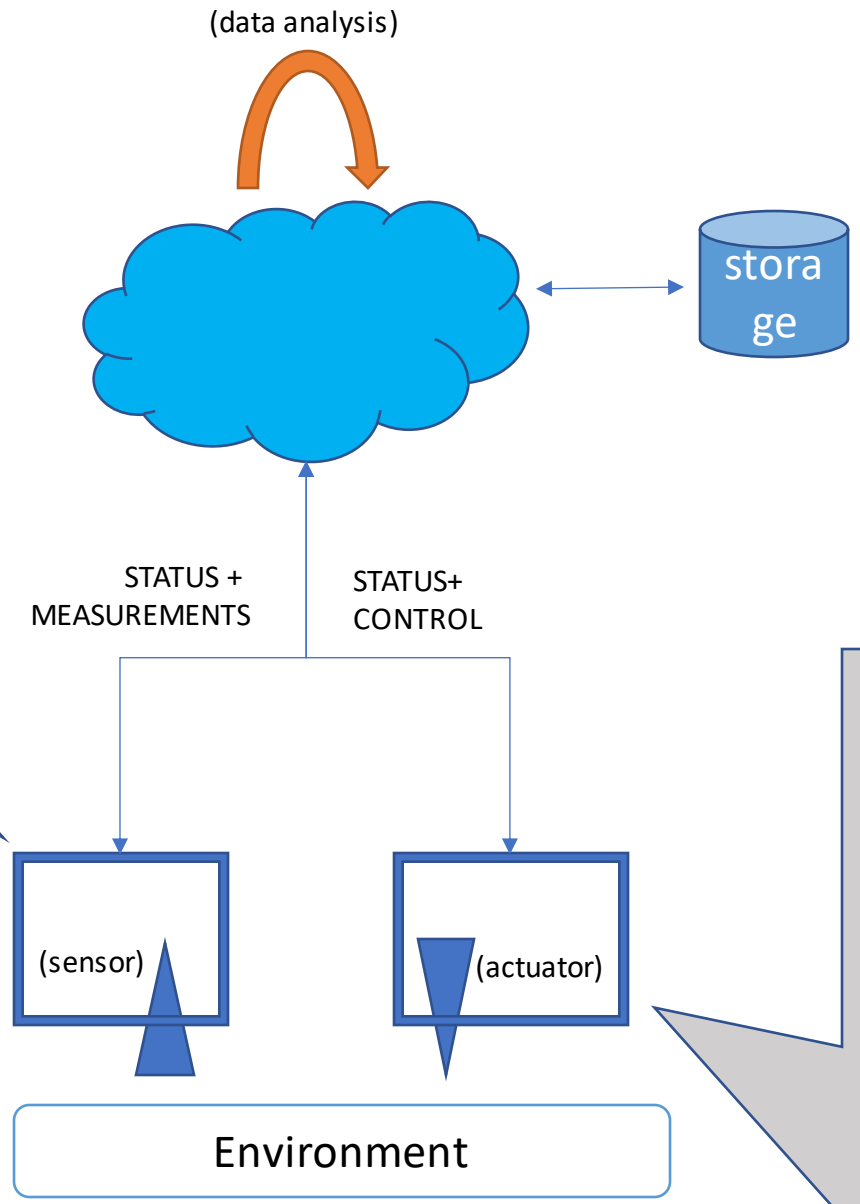
Co jest w środku:

- Sterowniki sensorów
- Sterowniki aktuatorór
- Algorytm kontroli w czasie rzeczywistym
- Sterownik pamięci zewnętrznej
- System plików
- Sterownik modemu
- Zdalne monitorowanie (status)

Co jest w środku:

- Sterowniki sensorów
- ~~Sterowniki aktuatorór~~
- ~~Algorytm kontroli w czasie rzeczywistym~~
- ~~Sterownik pamięci zewnętrznej~~
- ~~System plików~~
- Sterownik modemu
- Zdalne monitorowanie (status)

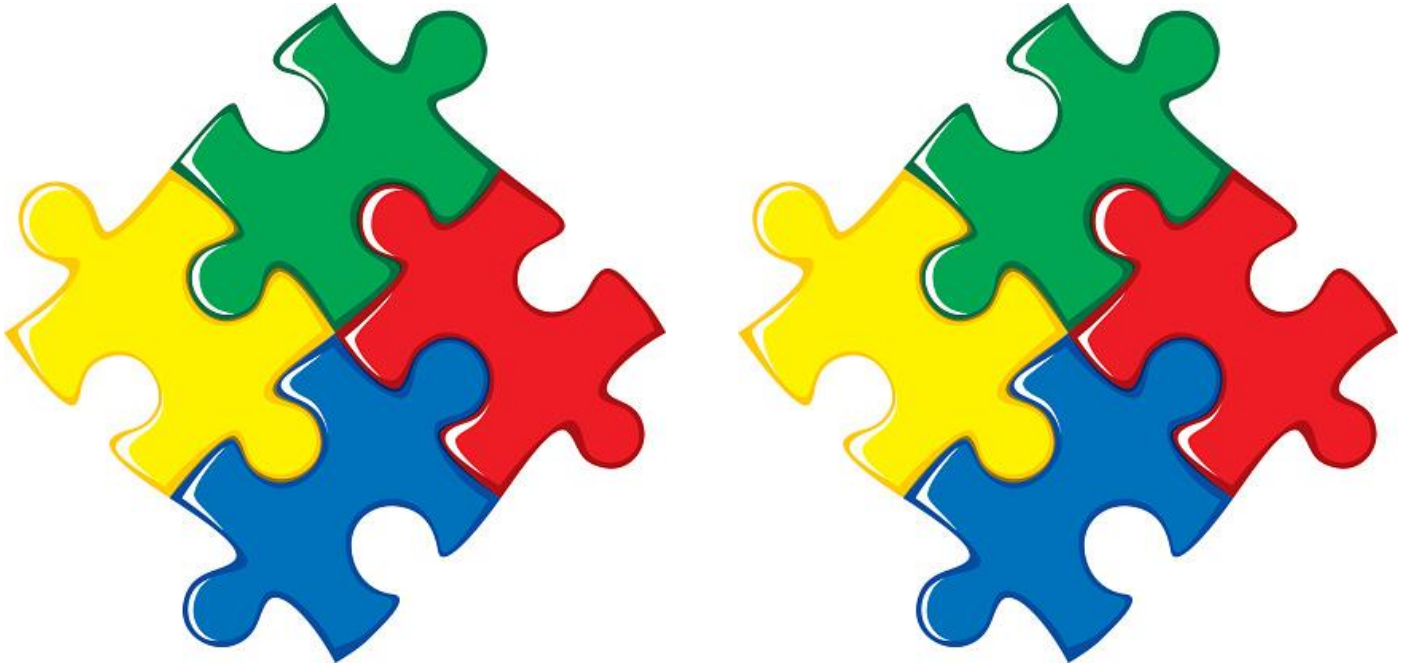
Pattern – Cloud Computing



Co jest w środku:

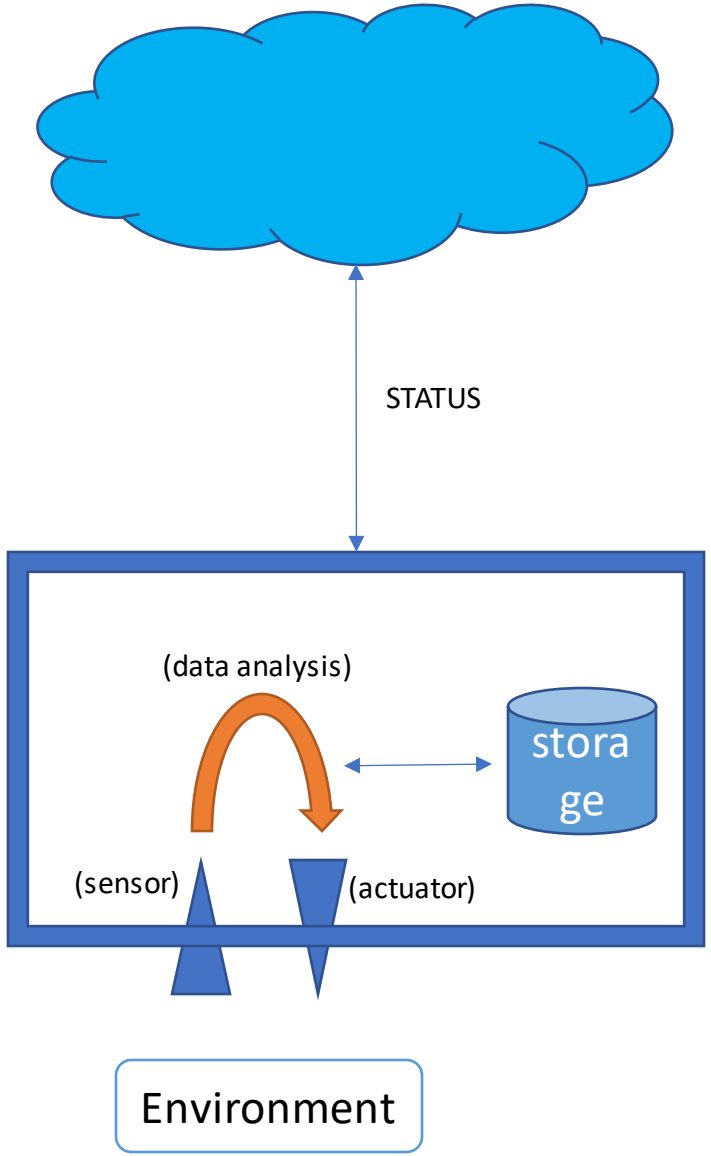
- ~~Sterowniki sensorów~~
- Sterowniki aktuatorór
- Algorytm kontroli w czasie rzeczywistym
- ~~Sterownik pamięci zewnętrznej~~
- ~~System plików~~
- Sterownik modemu
- Zdalne monitorowanie (status)

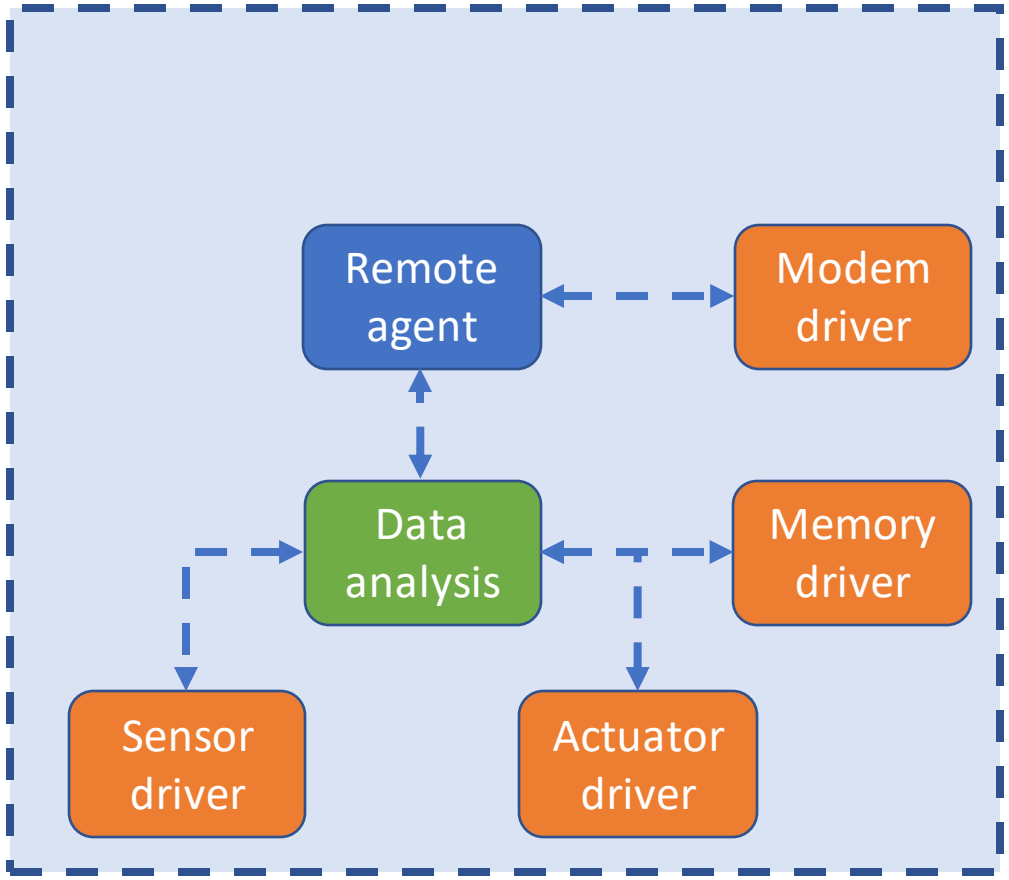
Modułowość i ponowne wykorzystanie oprogramowania

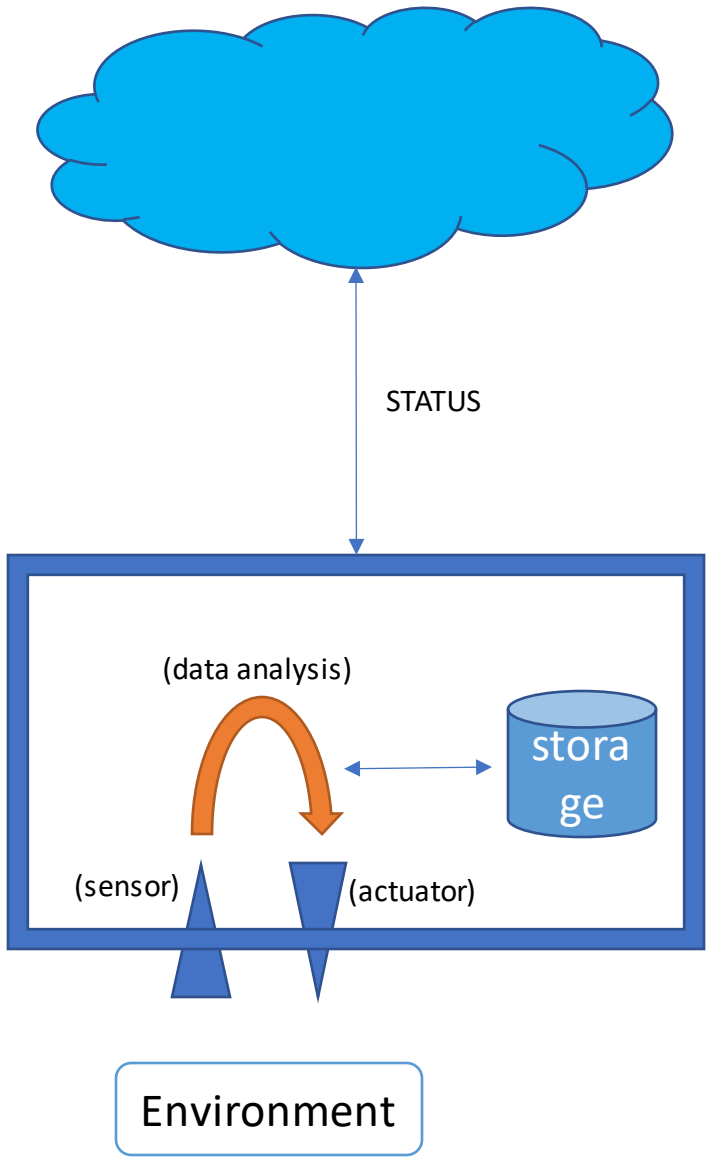


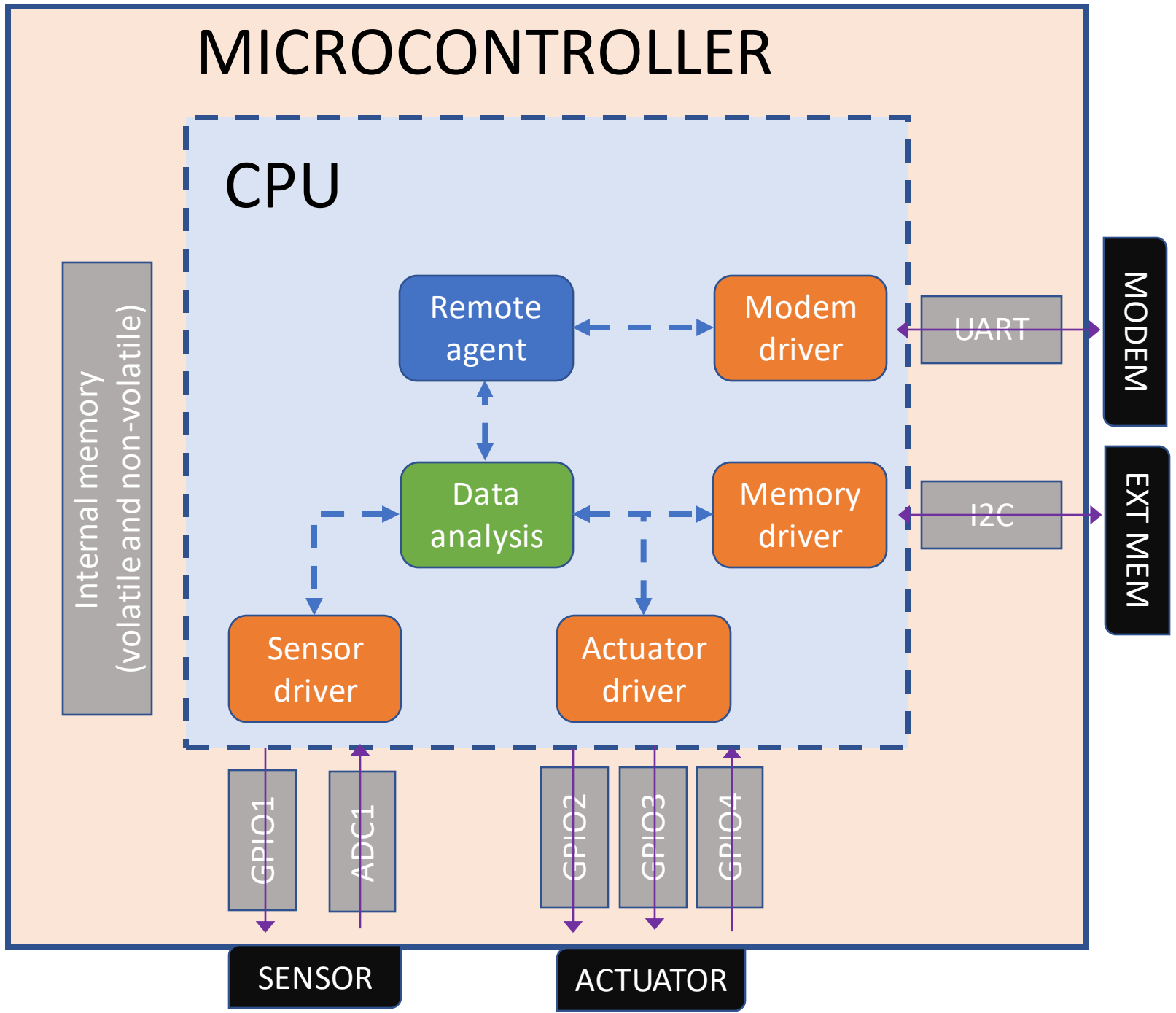
Tasks and resources

Service/Task	Peripheral access	Real-time response	Large memory	CPU calculations
Sterowniki sensorów	✓			
Sterowniki aktuatorów	✓	✓		
Sterownik modemu	✓			
Sterowniki radia	✓			
Networking stack				✓
Zdalny nadzór (status)				
Zdalne raportowanie (sensor)			✓	
Zdalna kontrola (aktuator)				


STATUS
(data analysis)
stora
ge
(sensor)
(actuator)
Environment


Remote
agent
Modem
driver
Data
analysis
Memory
driver
Sensor
driver
Actuator
driver


STATUS
(data analysis)
stora
ge
(sensor)
(actuator)
Environment


MICROCONTROLLER
CPU
Internal memory
(volatile and non-volatile)
Remote
agent
Modem
driver
UART
MODEM
Data
analysis
Memory
driver
I2C
EXT MEM
Sensor
driver
Actuator
driver
GPIO1
ADC1
GPIO2
GPIO3
GPIO4
SENSOR
ACTUATOR

Operating Systems

OS Theory

PROCESSES AND THREADS

MEMORY MANAGEMENT

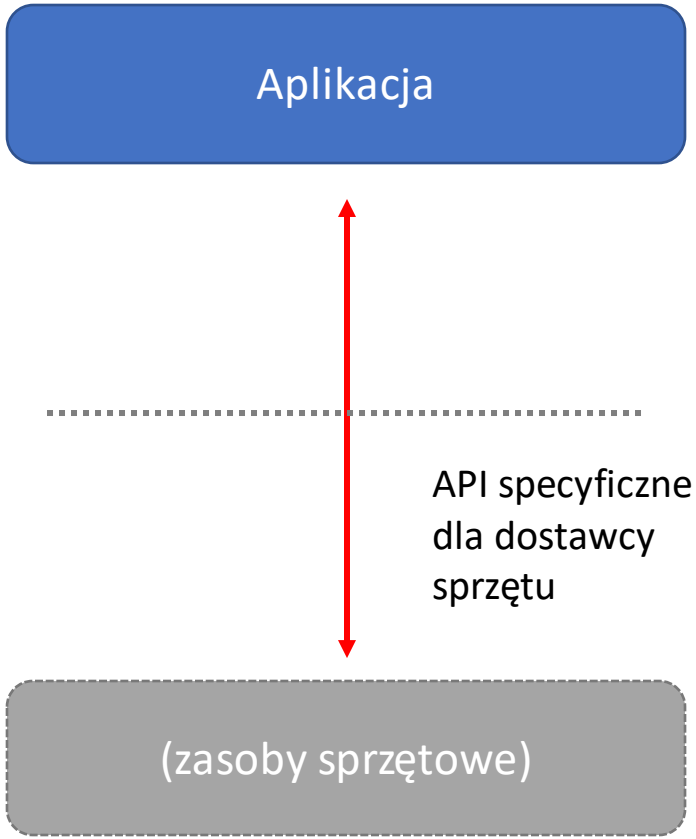
INPUT/OUTPUT

FILE SYSTEMS

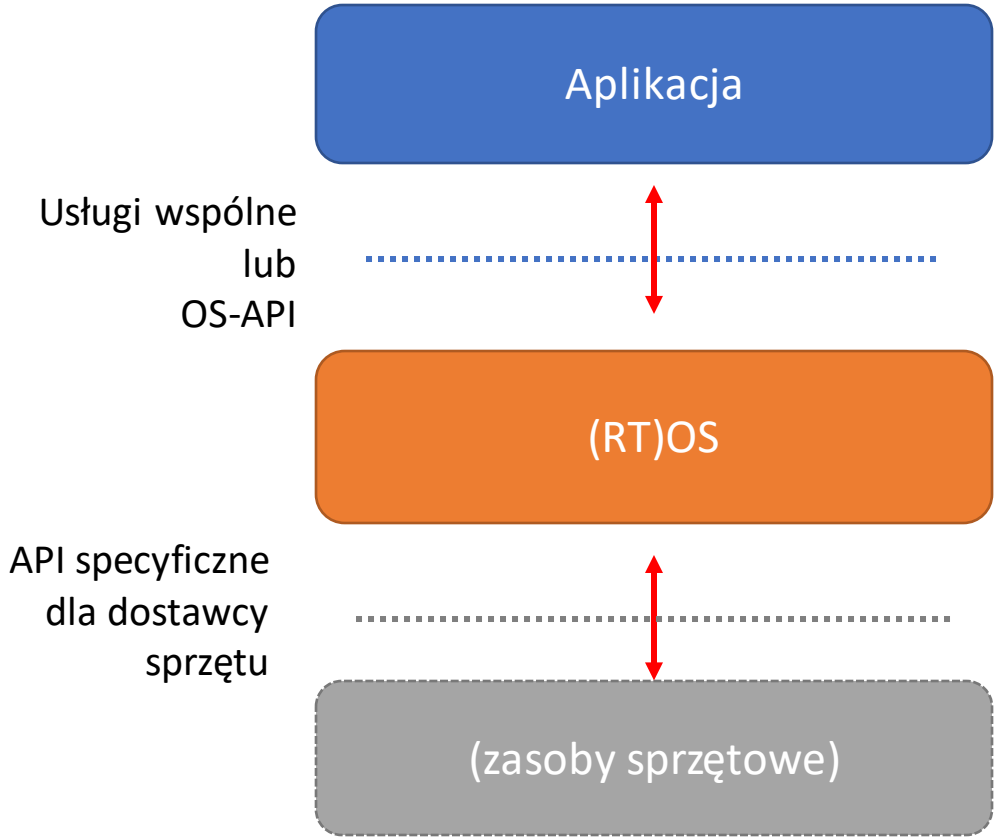
(Real-Time) Operating Systems

- System operacyjny (OS) to oprogramowanie systemowe, które zarządza sprzętem komputerowym i zasobami oprogramowania oraz zapewnia typowe usługi dla programów komputerowych.
- System operacyjny czasu rzeczywistego (RTOS) to system operacyjny (OS) przeznaczony do obsługi aplikacji czasu rzeczywistego, które przetwarzają dane w miarę ich pojawiania się, zazwyczaj bez opóźnień buforowania. Wymagania dotyczące czasu przetwarzania (w tym opóźnienia systemu operacyjnego) są mierzone w dziesiątych częściach sekundy lub krótszych odstępach czasu

Without OS

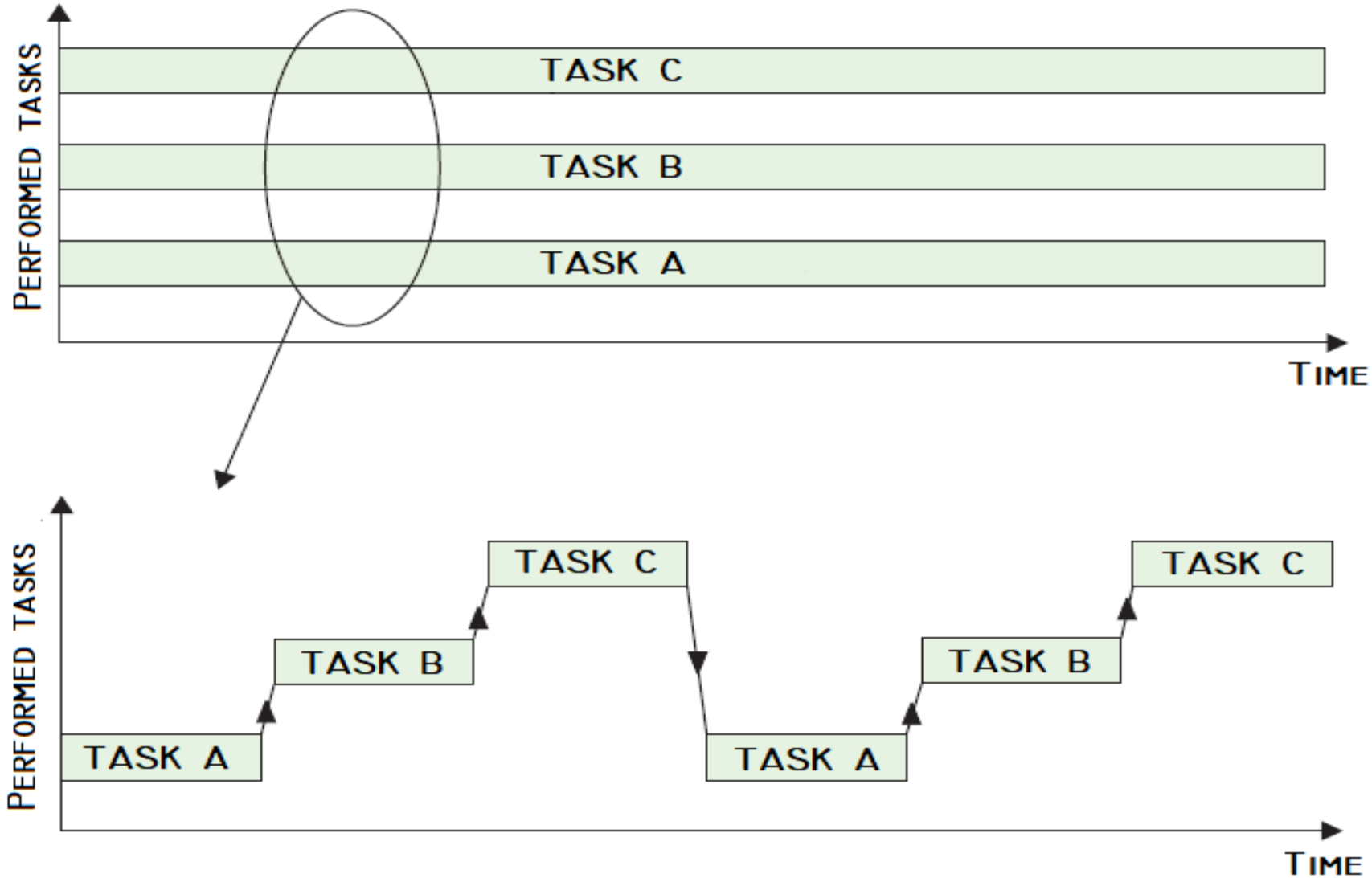


With OS



* Applications, processes, threads, threads

OS - Teoria


PERFORMED TASKS
TASK C
TASK B
TASK A
TIME
PERFORMED TASKS
TASK A
TASK B
TASK C
TASK A
TASK B
TASK C
TIME

Microkernel Basic Functionality

Planowanie wątków

(Threads Scheduling)

- Umożliwia wykonanie kilku wątków
- Izoluje przełączanie wątków od implementacji wątków
- Izoluje każdą implementację wątku

Komunikacja międzywątkowa

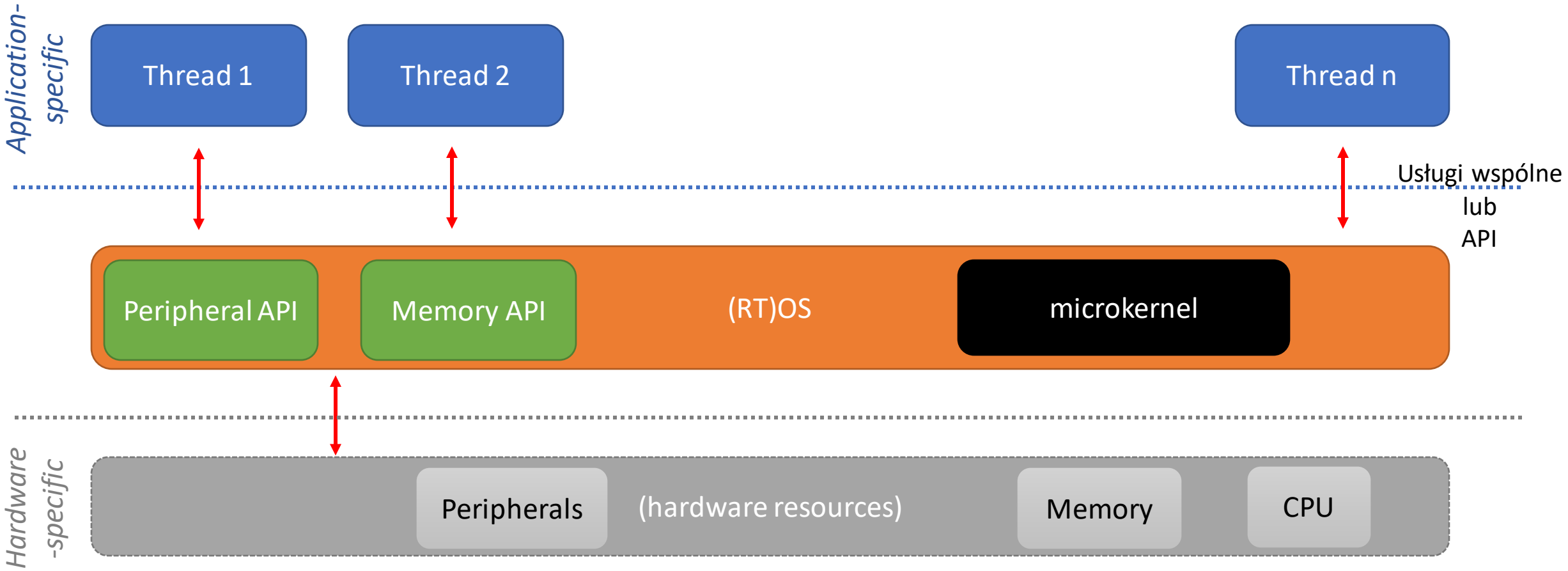
- Umożliwia przekazywanie wiadomości między wątkami
  - Kolejki
  - Kolejki priorytetowe
  - Semafory

RIOT's Microkernel (core)

```
 1009 24 maj   2016 atomic.c
 1542 19 sty   2016 bitarithm.c
 3862 24 maj   2016 c11_atomic.c
 4447 15 gru 16:24 kernel_init.c
 1383 19 sty   2016 lifo.c
11344 24 maj   2016 msg.c
 4151 24 maj   2016 mutex.c
 1992 24 maj   2016 panic.c
 2178 19 sty   2016 priority_queue.c
 3507 19 sty   2016 ringbuffer.c
 5482  8 cze   2016 sched.c
 6067  7 cze   2016 thread.c
 3751 24 maj   2016 thread_flags.c
```

+ Obsługa przerwań + Obsługa wyjątków (obsługiwane przez kontroler)

OS in IoT



* Applications, processes, threads, threads

ZADANIE

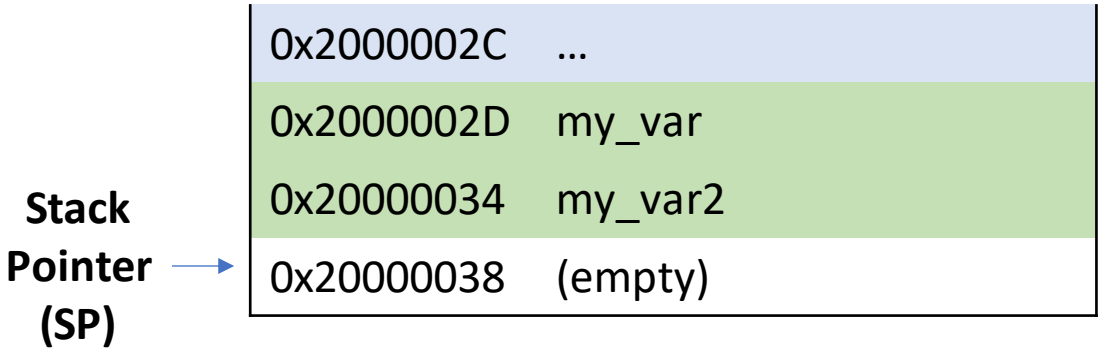
Importowanie projektu RIOT OS do Eclipe IDE

→ **Zadanie na komputerze**

→ **Instrukcja PDF**

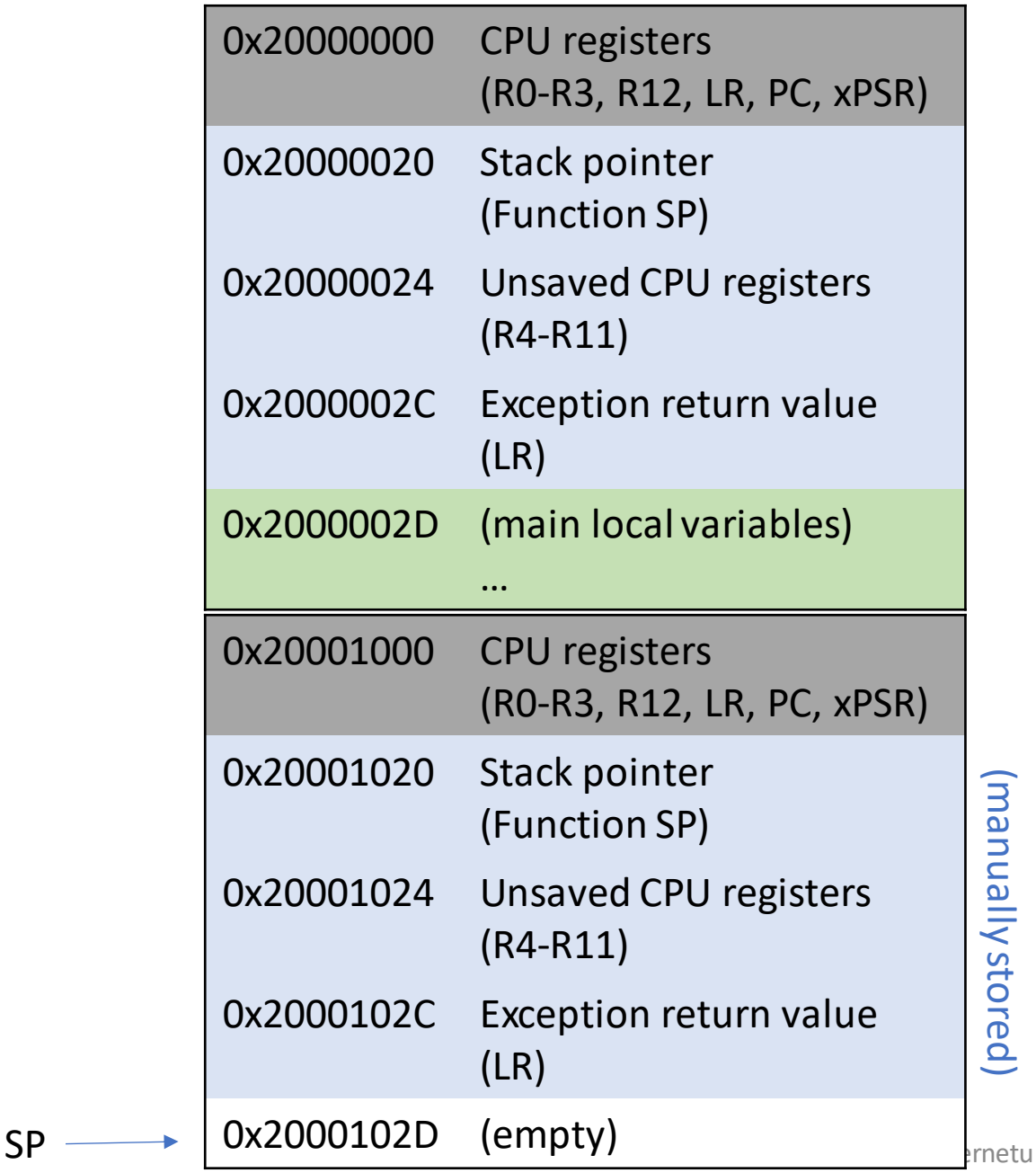
Wątki w RIOT-OS

Stack



Thread B Implementation

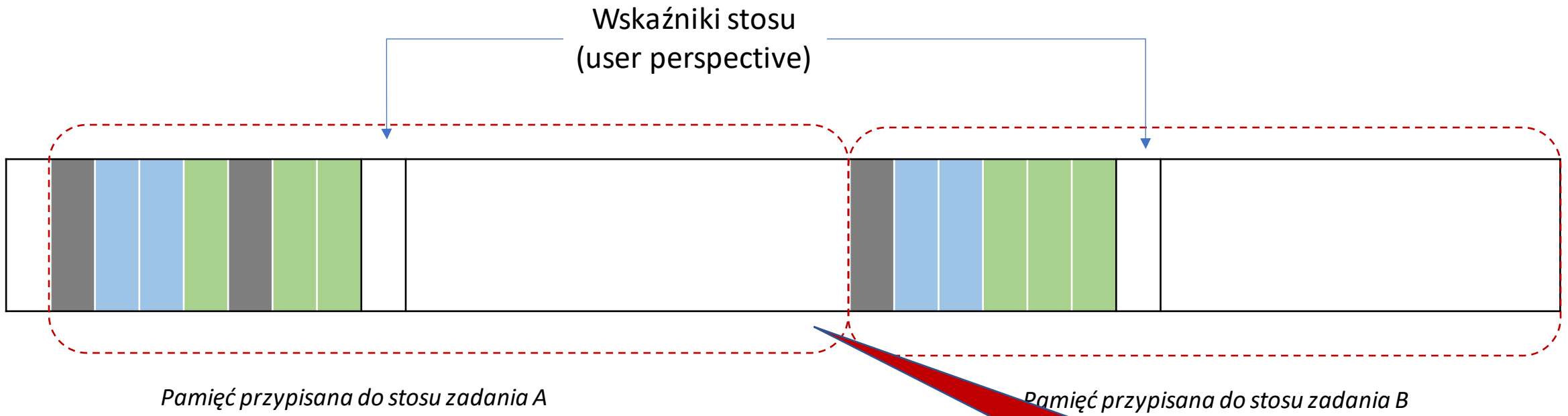
```
…
Void thread_B_main_loop(void){
    int my_var = 0;
    if(1){
        int my_var2 = 0;

        …
    }
    …
}
```



Thread B Implementation

```
…
Void thread_B_main_loop(void){
    …
    while(1){
        function_1_of_B_();
        …
    }
    …
}
```

OS z wywłaszczaniem



Za duży stos => strata pamięci

Za mały stos => uszkodzenie pamięci

W praktyce pamięć przypisana do stosów wątków może nie być ciągła (zmienne globalne mogą znajdować się pomiędzy)

Ile pamięci powinienem przypisać do mojego wątku?

- Zależy od
  - liczba funkcji ułożonych na stos
  - liczba zmiennych lokalnych używanych w każdej z tych funkcji
  - zastosowanie arytmetyki zmiennoprzecinkowej
- Wystrzegaj się
  - funkcji printf (i podobne funkcje formatowania ciągów)

W praktyce 500 KB – 3 KB na wątek

A view on RIOT's console

```
>
> ps
        pid | name                 | state    Q | pri | stack ( used) | location
          1 | idle                 | pending  Q |  15 |   256 (  220) | 0x20000598
          2 | main                 | running  Q |   7 |  1536 (  696) | 0x20000698
          3 | pktdump              | bl rx    _ |   6 |  1536 (  248) | 0x20003784
          4 | 6lo                  | bl rx    _ |   3 |  1024 (  296) | 0x20003d94
          5 | ipv6                 | bl rx    _ |   4 |  1024 (  260) | 0x20001970
          6 | ieee802154_control   | bl rx    _ |   4 |  1024 (  440) | 0x200049a4
          7 | udp                  | bl rx    _ |   5 |  1024 (  272) | 0x2000459c
          8 | coap                 | bl rx    _ |   6 |  1536 (  556) | 0x20001344
          9 | ezradio2             | bl rx    _ |   2 |  1024 (  552) | 0x20000cf0
            | SUM                  |            |     |  9984 ( 3540)
>
```

Całkowita i wykorzystana pamięć w stosie

Początek stosu w pamięci

Dziewięć (9) wątków

Creating a thread

```
kernel_pid_t thread_pid = thread_create(
            pointer_to_thread_stack_start,
            size_of_stack,
    thread_priority,
    thread_flags,
    thread_function_name,
    thread_function_arguments,
    thread_id);
```

```
void * thread_function_name(void* thread_function_arguments){
            while(1) {
                        …
            }
            return NULL;
}
```

ZADANIE

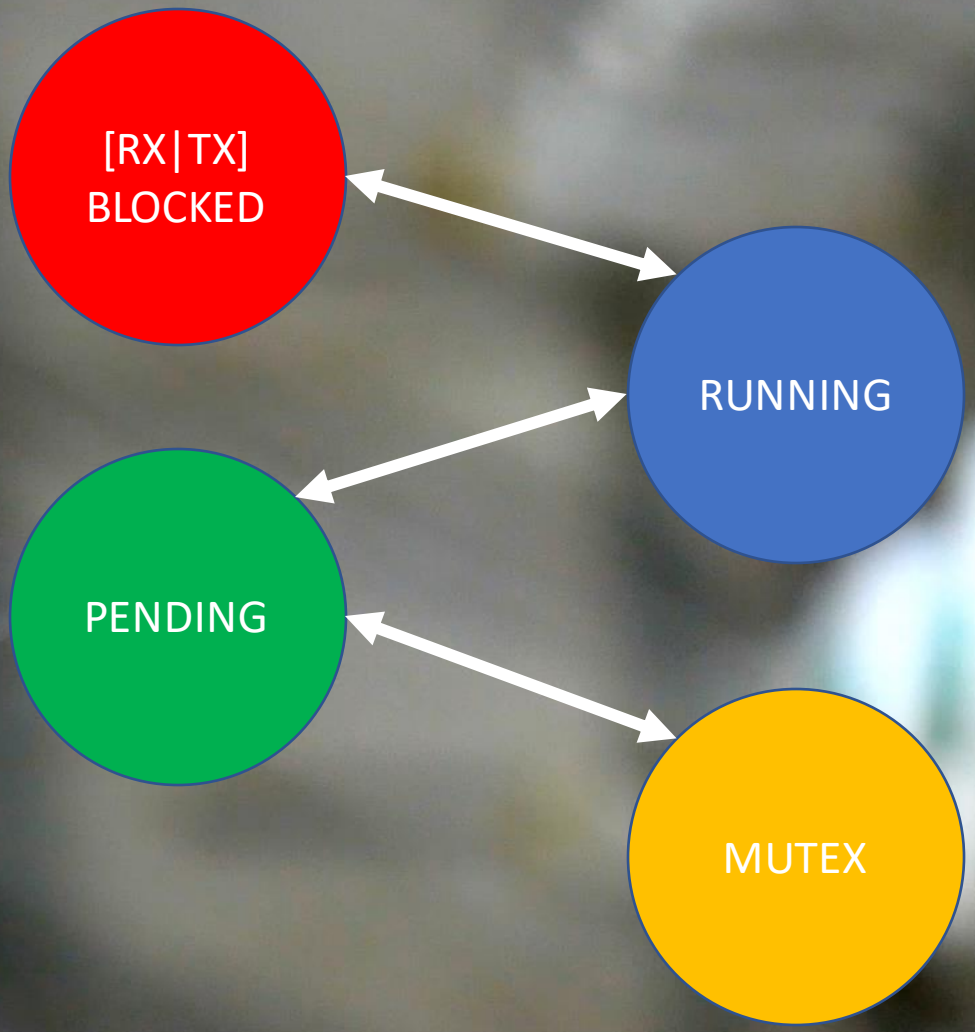
Tworzenie wątków

→ **Zadanie na komputerze**

→ **Instrukcja PDF**

Threads Scheduling

Thread's state



Stan wątku nie może być zmieniony bezpośrednio przez API wysokiego poziomu…

DISCLAIMER: These are only the most used ones in RIOT-OS

Klasyfikacja planowania OS

~~Planowanie współdzielcze~~

- Każdy wątek decyduje, kiedy oddać kontrolę nad wykonaniem do innych wątków
- Po przerwaniu system wraca do tego samego wątku
- Może spowodować zagłodzienie wątku
  - Źle zaimplementowany wątek może się zawieść i nigdy nie dać kontroli

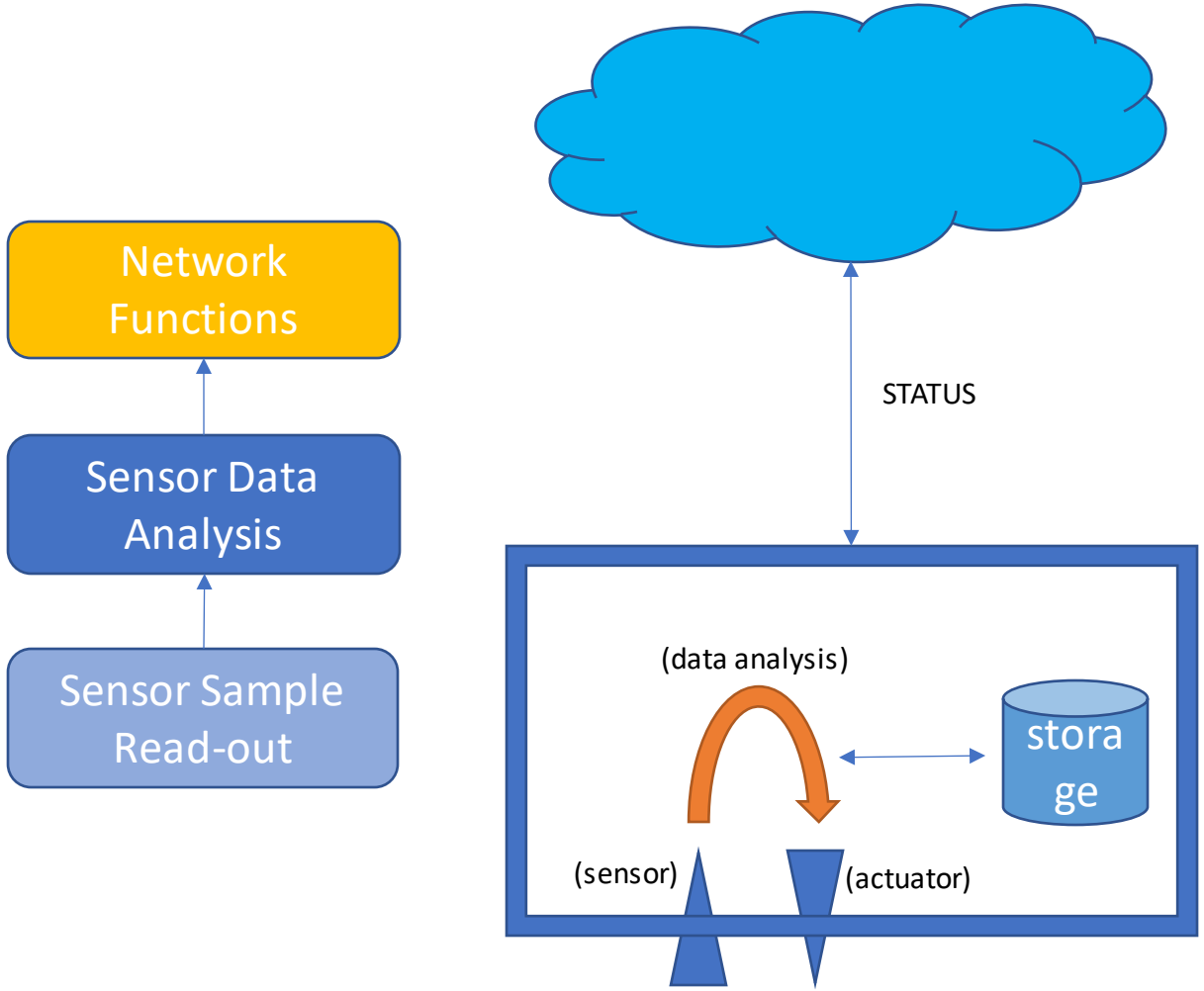
Planowanie z wywłaszczaniem

- Wykonanie wątku może zostać przejęte z bieżącego wątku i przekazane innemu przez system operacyjny w dowolnym momencie,
- Po przerwaniu system może powrócić do innego wątku
- OS **MUSI** przechowywać stos wywołań każdego wątku przed wywłaszczaniem (przełączanie)
- Zwykle wymaga więcej pamięci i energii
  - Każdy wątek ma osobny stos pamięci, który zwykle jest przydzielany statycznie

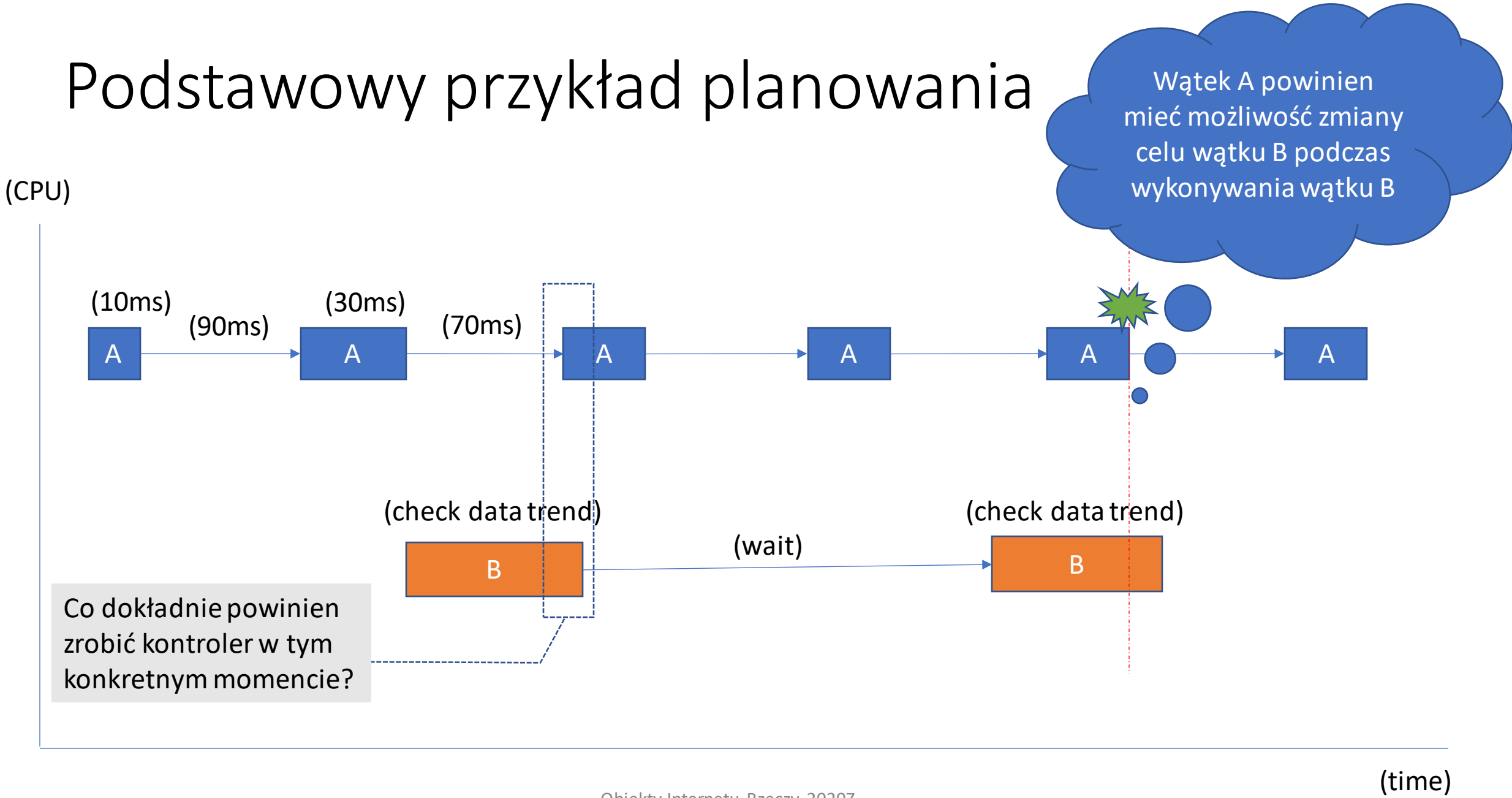
Podstawowy przykład planowania – scenariusz

Proces pomiarowy pierścienia Micromole

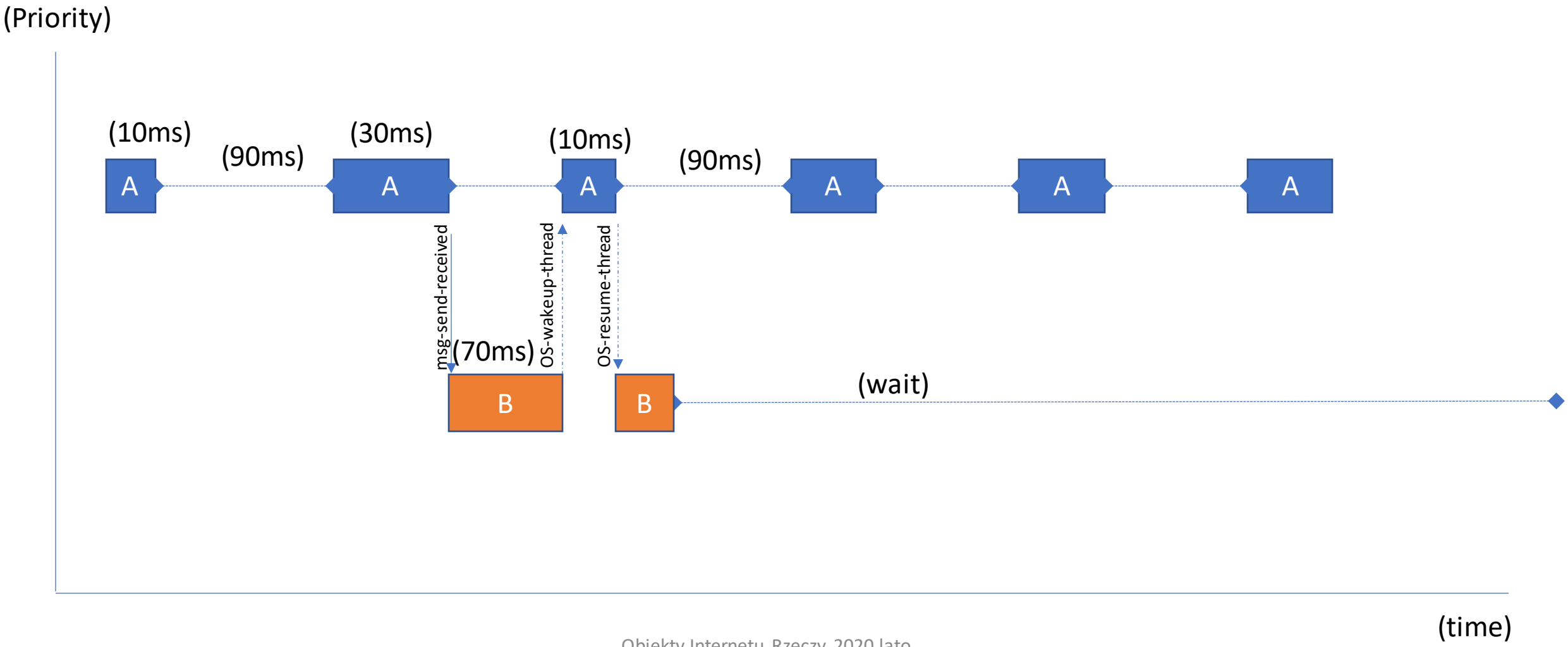
- Dwa wątki:
  - Wątek A: Odczyt danych z sensora
    - Odczyt próbki co 100 ms
    - Pomiar zawiera się między 10 a 30 ms
    - Wysyła wartość pomiaru do Wątku B
  - Wątek B: Analiza danych z sensora
    - Sprawdzenie trendu z ostatnich N próbek trwa 80 ms
    - Jeśli ostatnia próbka jest poza trendem, wyślij powiadomienie



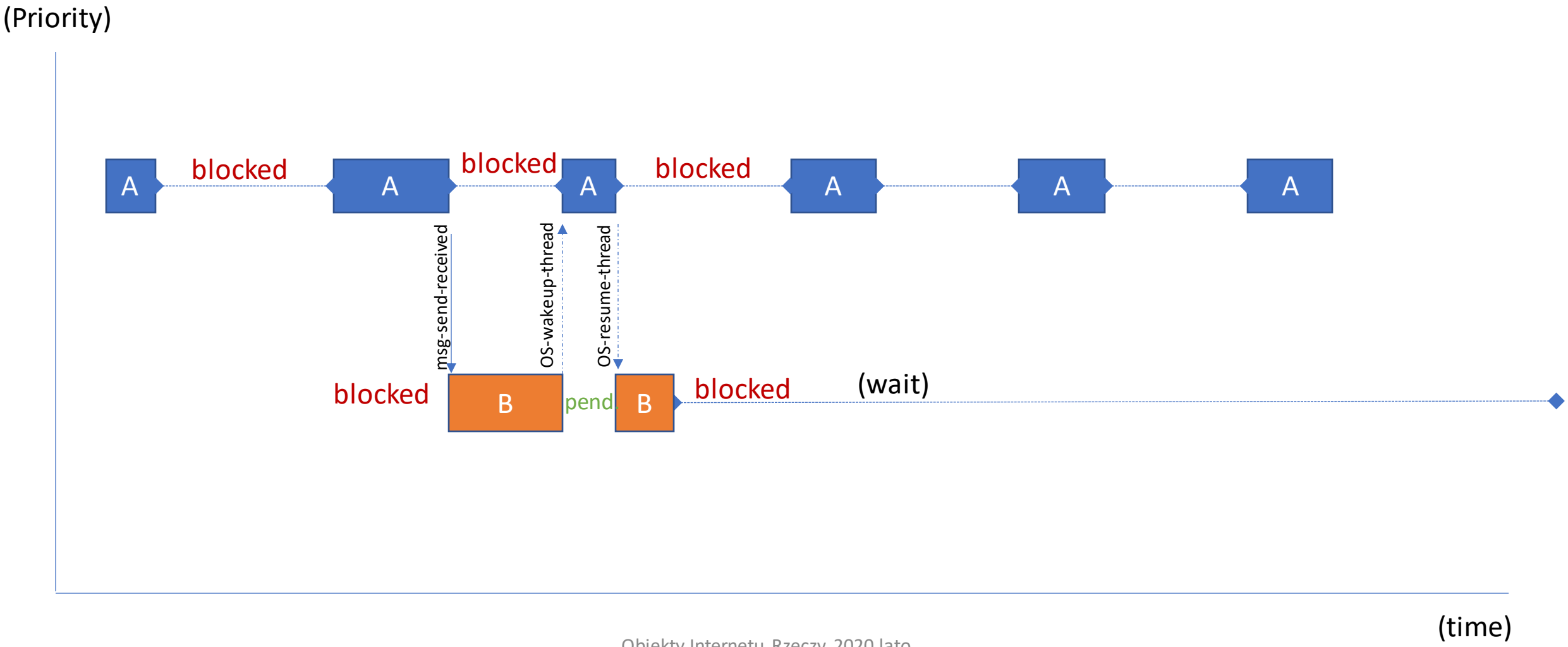
Podstawowy przykład planowania



(Oparte na priorytetach) Planowanie z wywłaszaniem

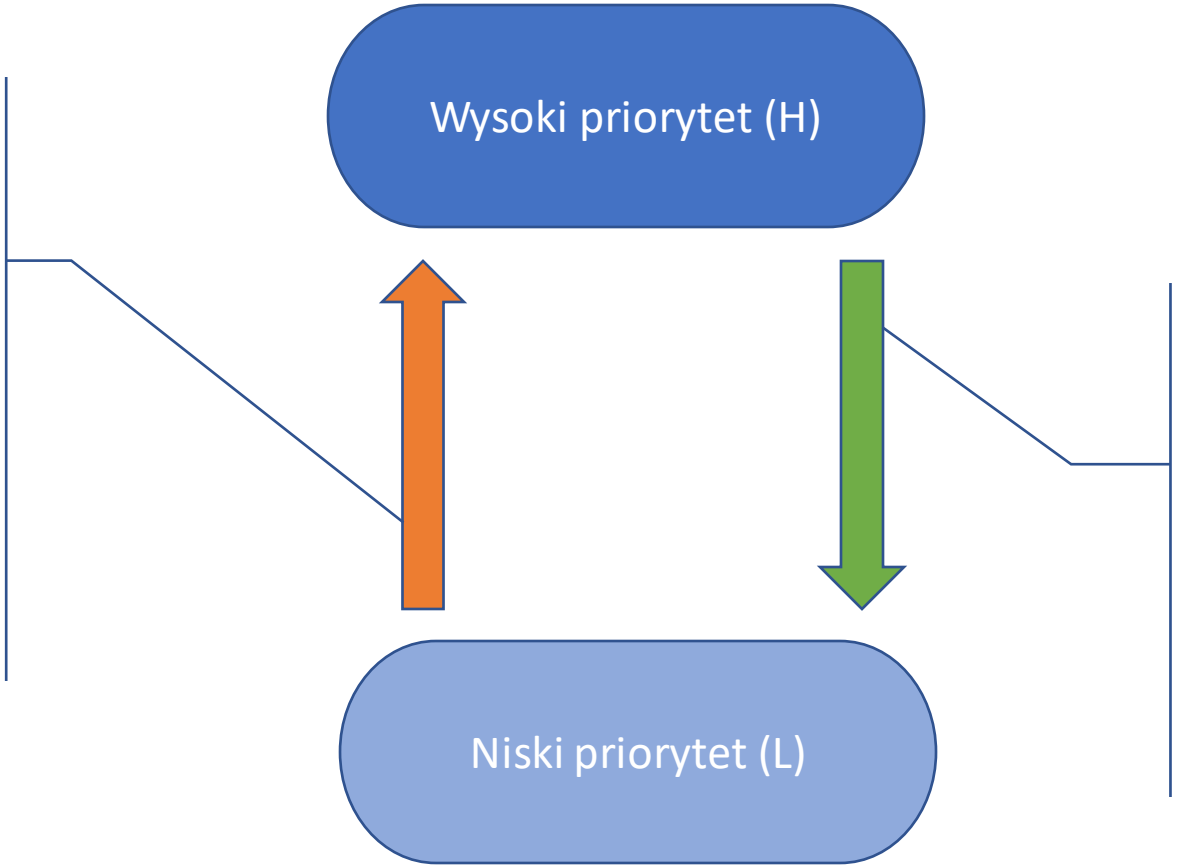


(Oparte na priorytetach) Planowanie z wywłaszaniem



Zasady przełączania kontekstu …

- PRZED: wątek H jest zablokowany, wątek L jest wykonywany
- W TRAKCIE: wątek L (lub ISR) bezpośrednio lub pośrednio odblokowuje H
- PO: wątek L oczekuje na wykonanie, wątek H jest wykonywany



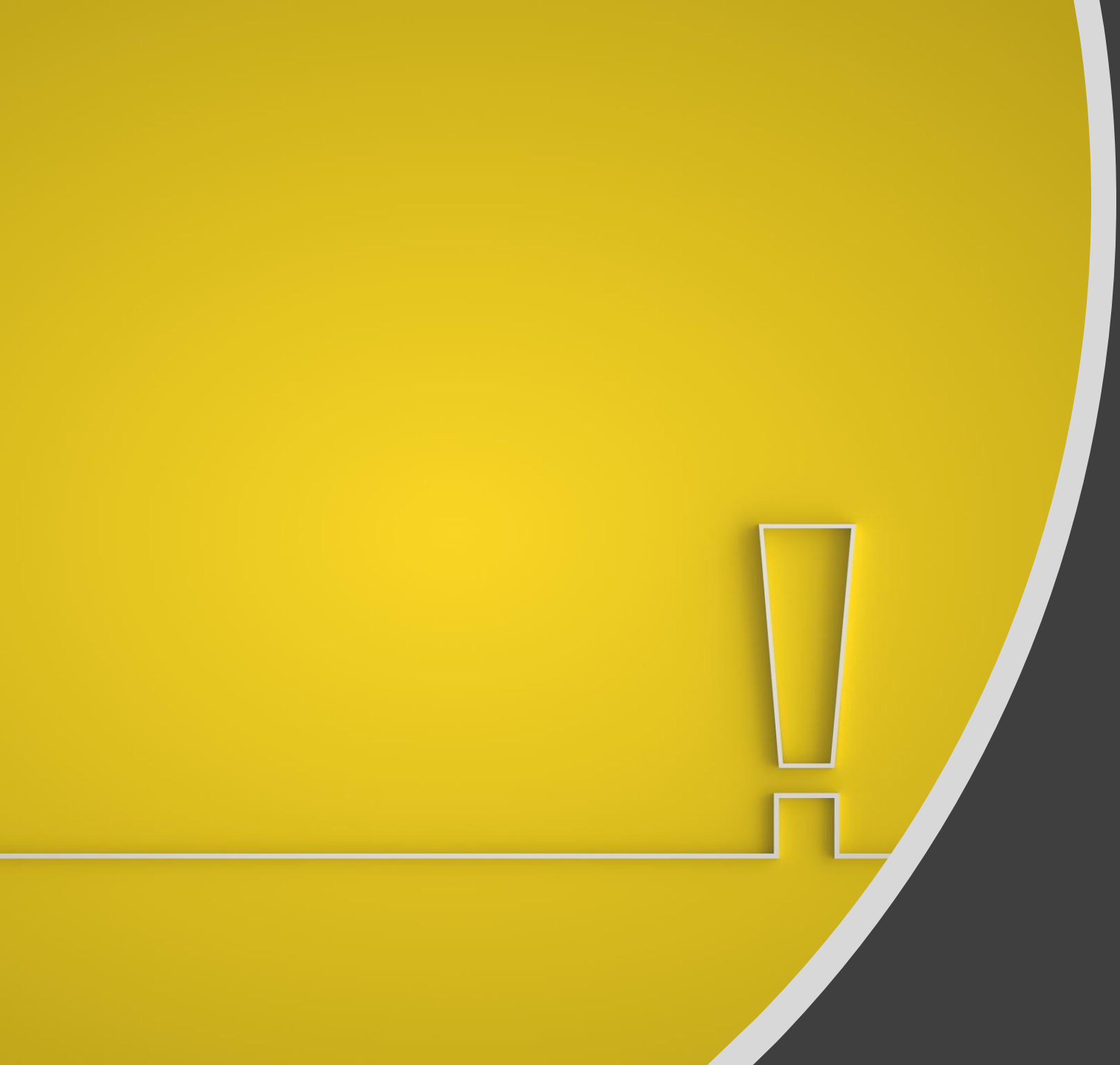
- PRZED: wątek H jest uruchomiony, wątek L oczekuje
- W TRAKCIE: wątek H zostaje zablokowany
- PO: wątek H jest zablokowany, wątek L jest wykonywany

A view on RIOT’s console

Priorytet

```
>
> ps
        pid | name                 | state    Q | pri | stack ( used) | location
          1 | idle                 | pending  Q |  15 |   256 (  220) | 0x20000598
          2 | main                 | running  Q |   7 |  1536 (  696) | 0x20000698
          3 | pktdump              | bl rx    _ |   6 |  1536 (  248) | 0x20003784
          4 | 6lo                  | bl rx    _ |   3 |  1024 (  296) | 0x20003d94
          5 | ipv6                 | bl rx    _ |   4 |  1024 (  260) | 0x20001970
          6 | ieee802154_control   | bl rx    _ |   4 |  1024 (  440) | 0x200049a4
          7 | udp                  | bl rx    _ |   5 |  1024 (  272) | 0x2000459c
          8 | coap                 | bl rx    _ |   6 |  1536 (  556) | 0x20001344
          9 | ezradio2             | bl rx    _ |   2 |  1024 (  552) | 0x20000cf0
            | SUM                  |            |     |  9984 ( 3540)
>
```

9 wątków

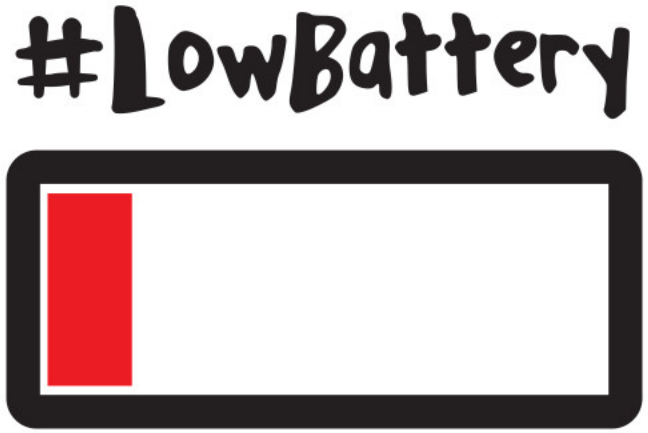


Interruptions

Wątek A – Unikaj używania czujnika przy niskiej mocy

```
while(1){
// Function is blocking (10 ms)
  value = read_sample();
  msg_t msg;
  msg.type = MSG_TYPE_MEASUREMENT;
  msg.content.value = value;
  msg_send(&msg, pid_thread_B);
  thread_sleep(90ms);
}
```

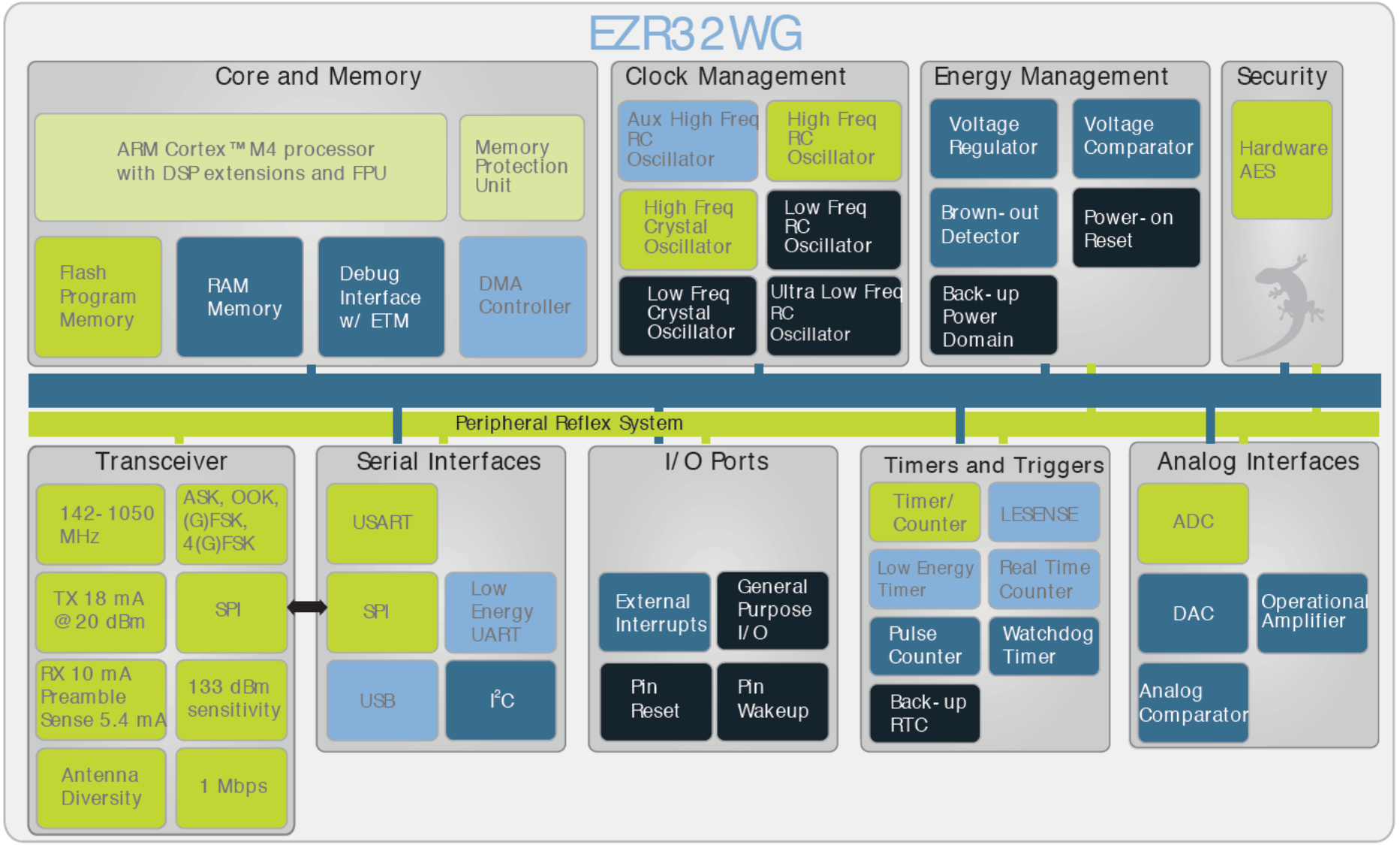
```
while(1){
  if(GPIO->PORTA == 0){
//   Function is blocking (10 ms)
    msg_t msg;
    msg.type = MSG_TYPE_MEASUREMENT;
    msg.content.value = value;
    msg_send(&msg, pid_thread_B);
  }
  else{
    notify_low_voltage();
  }
  thread_sleep(90ms);
}
```



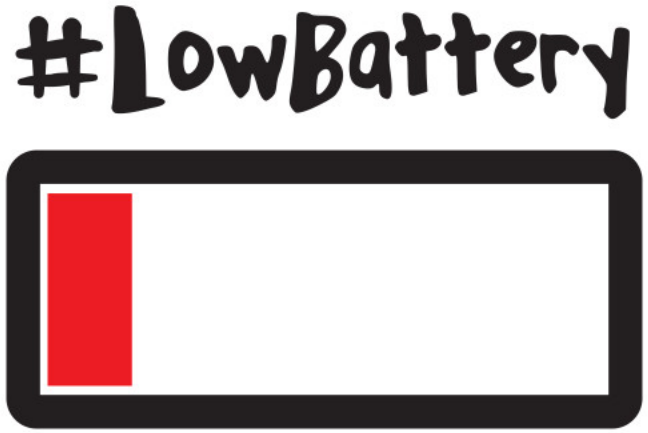
Odczytuje wartość pinu A1 kontrolera

Jeśli pin jest ustawiony nisko podczas uśpienia wątku, powiadomienie zostanie opóźnione o max. 90 ms

Peripherals in EZR32WG



Wątek A – Unikaj używania czujnika przy niskiej mocy

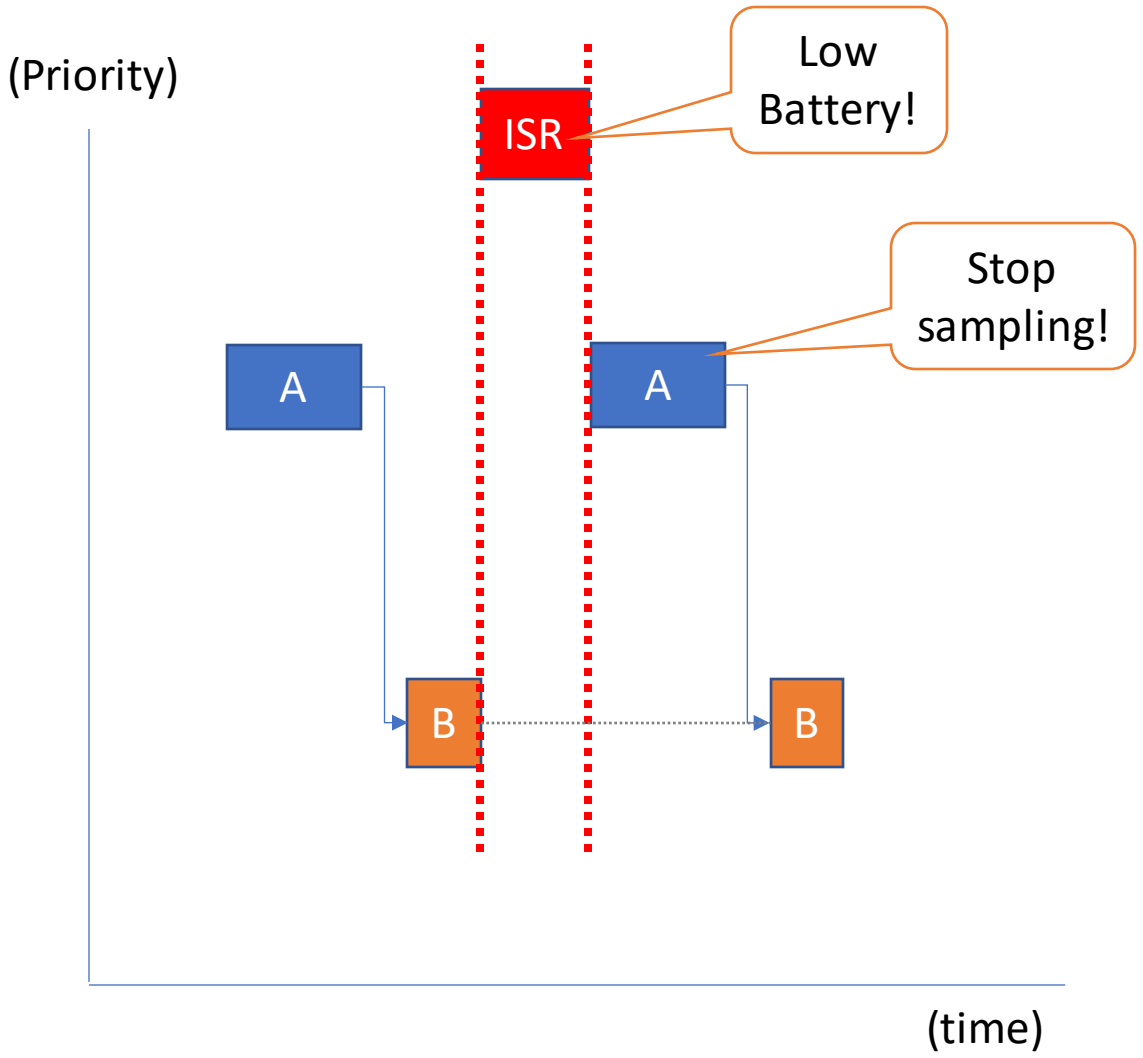



```
while(1){
  if(is_battery_low == 0){
  // Function is blocking (10 ms)
    value = read_sample();
    msg_t msg;
    msg.type = MSG_TYPE_MEASUREMENT;
    msg.content.value = value;
    msg_send(&msg, pid_thread_B);
  }
  thread_sleep(90ms);
}
```

```
void isr_gpio(void){
  is_battery_low = GPIO->PORTA;
  if(is_battery_low == 1){
    notify_low_voltage();
  }
}
```

Ta funkcja jest wywoływana automatycznie, gdy tylko kontroler wykryje zmianę stanu pinu w sprzęcie

ISR in pre-emptive OS

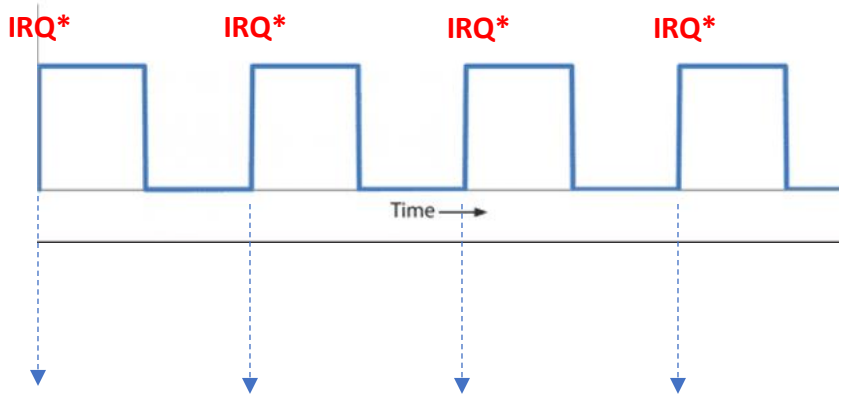



```
while(1){
  if(is_battery_low == 0){
  // Function is blocking (10 ms)
    value = read_sample();
    msg_t msg;
    msg.type = MSG_TYPE_MEASUREMENT;
    msg.content.value = value;
    msg_send(&msg, pid_thread_B);
  }
  thread_sleep(90ms);
}
```

```
void isr_gpio(void){
  is_battery_low = GPIO->PORTA;
  if(is_battery_low == 1){
    notify_low_voltage();
  }
}
```

System Timer and Thread Scheduling

A Simple OS Thread Scheduler



.e.g, Linux…

```
void isr_timer(void){
  thread_t* next_thread = select_next_pending_thread();
  save_current_thread_context();
  run_thread(next_thread);
}
```

Inne przykłady procesów sterowanych przerwaniami

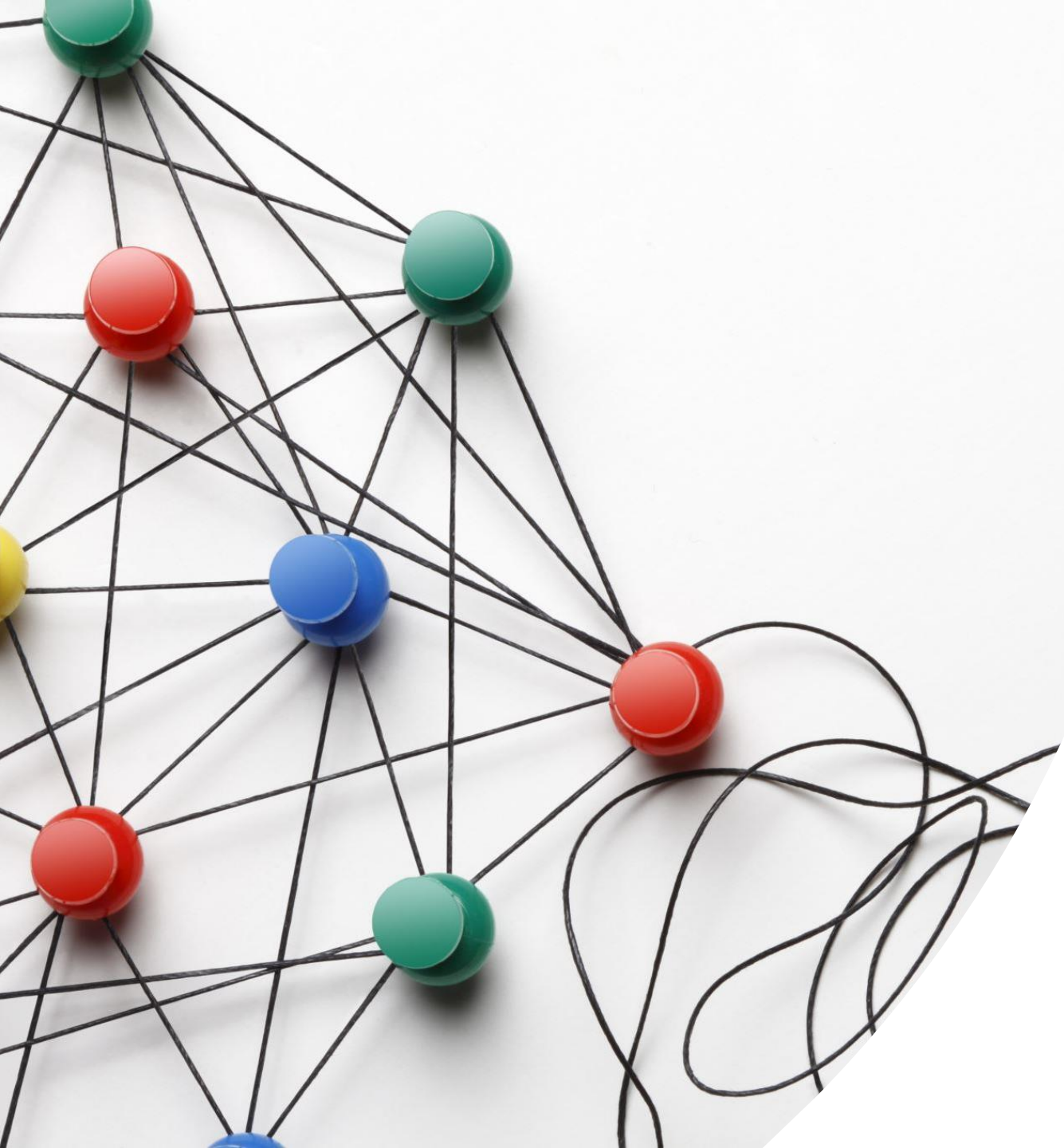
- Odczytywanie wejścia UART (np. wejście konsoli - klawiatura)
- Wysyłanie danych do UART (np. wyjście z konsoli – wyświetlenie na ekranie)
- Pobieranie próbki analogowej za pomocą ADC
- Wysyłanie/odbieranie danych przez magistralę komunikacyjną (I2C, SPI, USB, CAN itp.)
- Wykonywanie szyfrowania/odszyfrowywania AES-128 lub AES-192
- Generuj sygnału pulsującego (PWM) na pinie uC
- Odczyt/zapis z pamięci RAM za pomocą DMA
- …

ZADANIE

Obsługa przerwania, odmierzanie czasu

→ **Zadanie na komputerze**

→ **Instrukcja PDF**



Synchronizacja Między Wątkami

Service/Task	GPIO	SPI	UART	Dostęp do pamięci	Dostęp do CPU
Sterowniki sensorów	✓	✓			
Sterowniki aktuatorów	✓	✓			
Sterownik modemu	✓	✓	✓	✓	✓
Sterowniki radia	✓	✓			✓
Networking stack					✓

Mechanizmy do komunikacji pomiędzy zadaniami (lub przerwaniami i zadaniami) oraz do zabezpieczania zasobów mikrokontrolera

1. **Semafory**
2. **Muteksy**
3. **Wiadomości**
4. **Kolejki**

Semafory

void sema_create (**sema_t** *sema, **unsigned int** value)

void sema_destroy **(sema_t** *sema)

static unsigned sema_get_value (**const sema_t** *sema)

int _sema_wait_ztimer (**sema_t** *sema, **int** block,
    ztimer_clock_t *clock, **uint32_t** timeout)

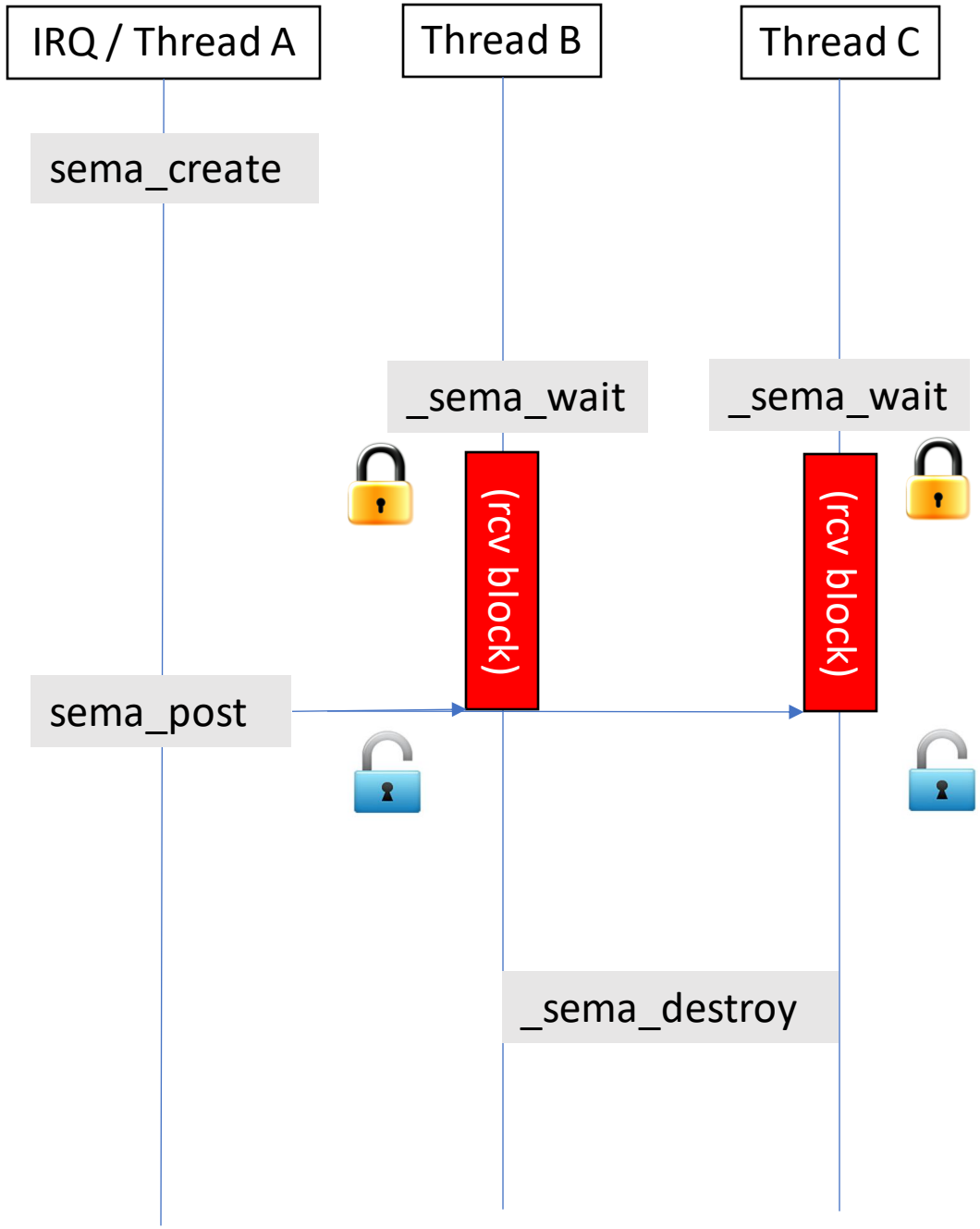
static int sema_wait (**sema_t** *sema)

static int sema_try_wait (**sema_t** *sema)

static int sema_wait_timed (**sema_t** *sema, **uint64_t** timeout)

static int sema_wait_timed_ztimer (**sema_t** *sema,
    ztimer_clock_t *clock, **uint32_t** timeout)

int sema_post (**sema_t** *sema)



Muteksy

static void mutex_init (**mutex_t** *mutex)

static mutex_cancel_t mutex_cancel_init (mutex_t *mutex)

int mutex_trylock_ffi (mutex_t *mutex)

static int mutex_trylock (mutex_t *mutex)

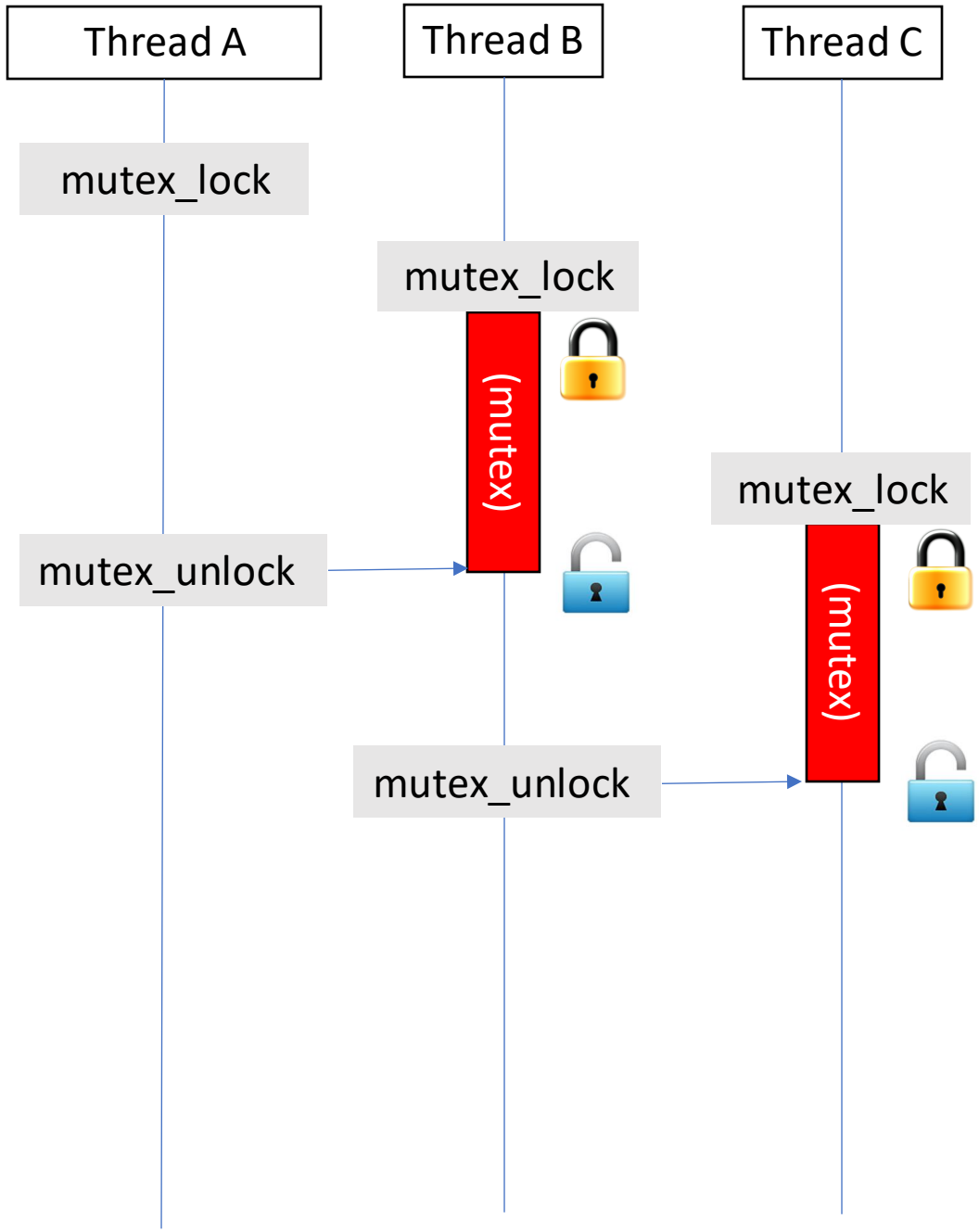
void mutex_lock (mutex_t *mutex)

int mutex_lock_cancelable (mutex_cancel_t *mc)

void mutex_unlock (mutex_t *mutex)

void mutex_unlock_and_sleep (mutex_t *mutex)

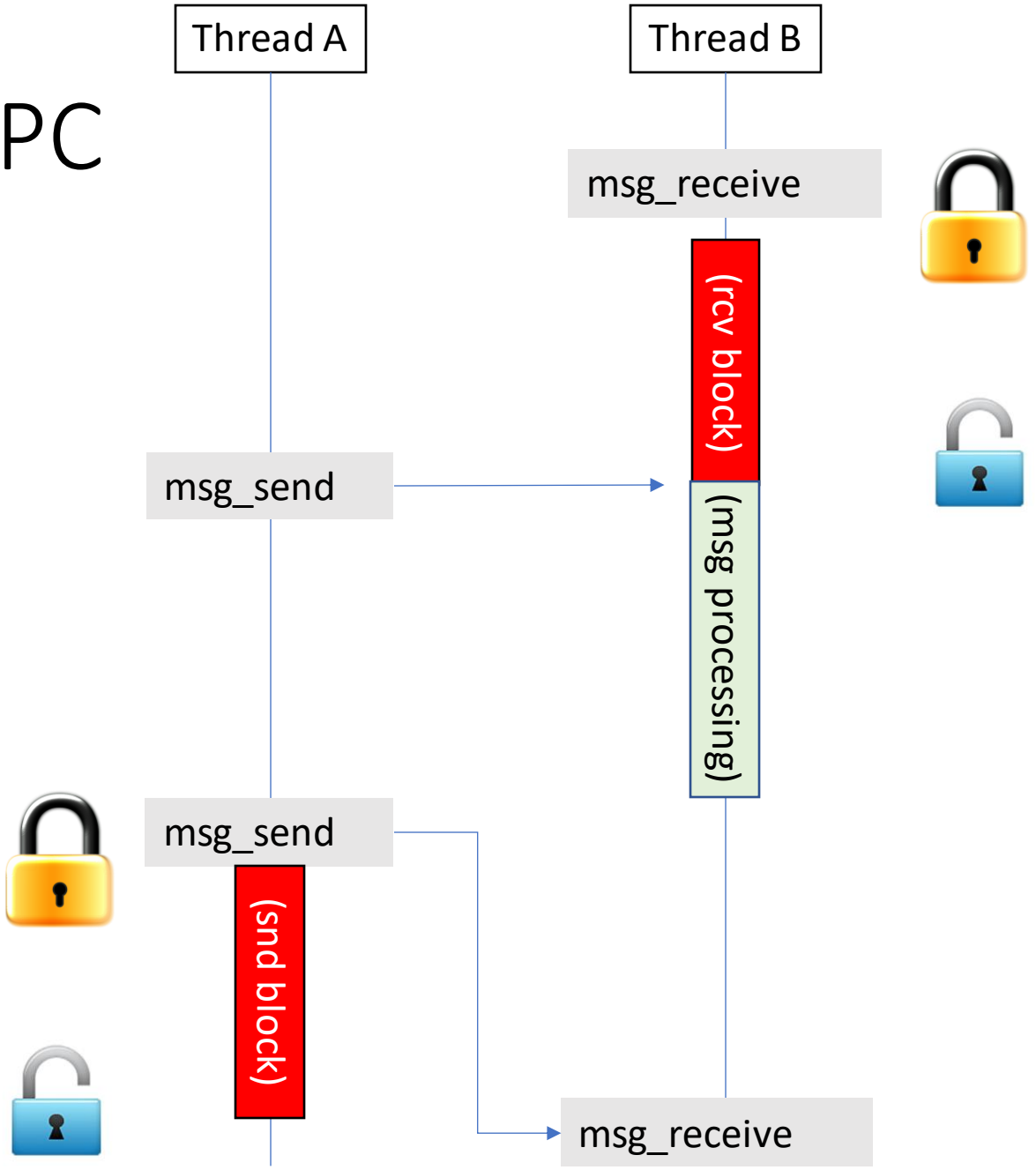
void mutex_cancel (mutex_cancel_t *mc)



Synchronizacja blokująca IPC

```
int msg_send(msg_t *m,  kernel_pid_t
    target_pid
```

```
int msg_receive(msg_t *m)
```



ZADANIE

Mutexy

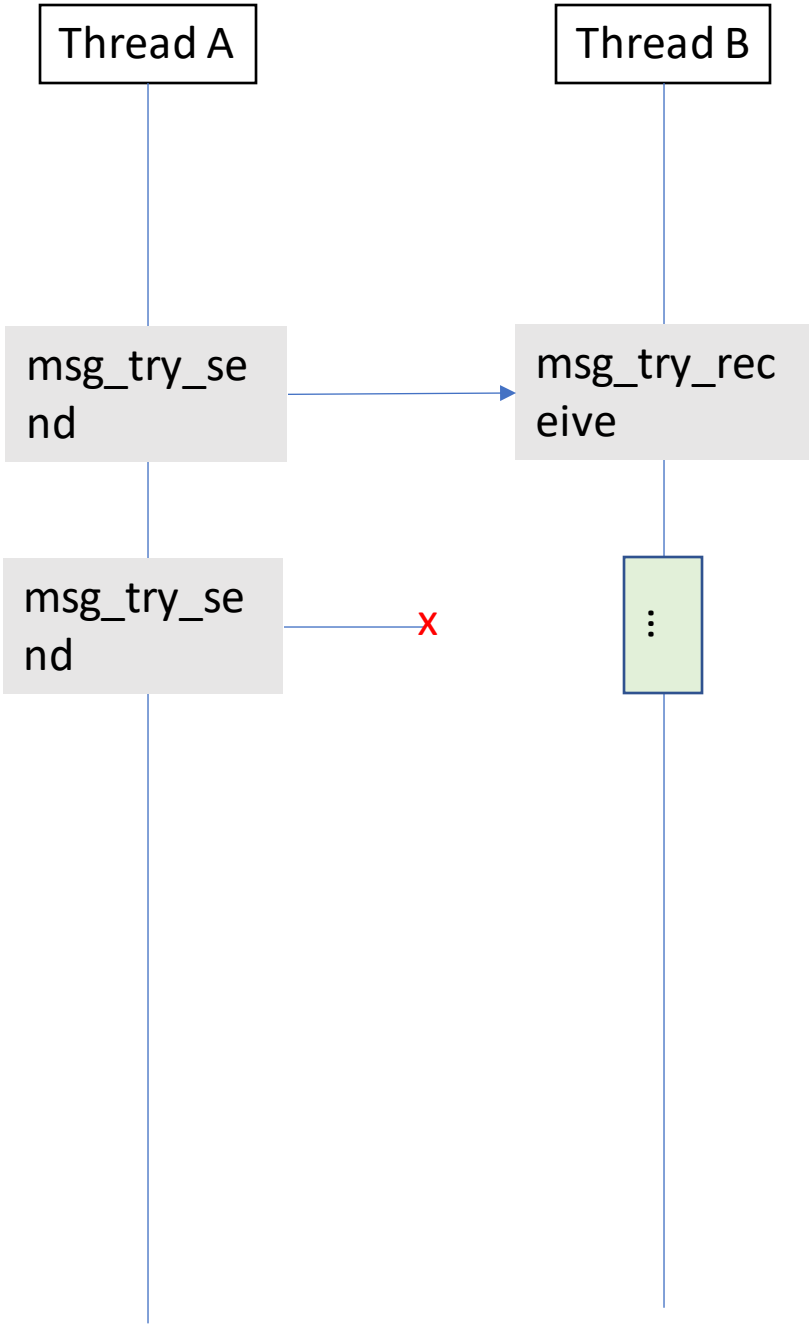
→ Zadanie na komputerze

→ Instrukcja PDF

Synchronizacja nieblokująca IPC

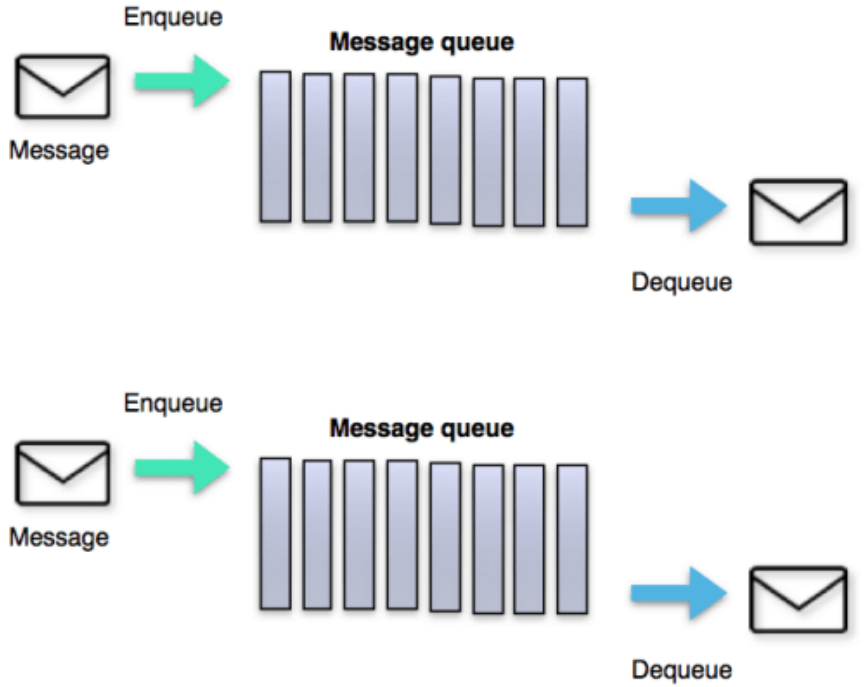
```
int msg_try_send(msg_t *m,  kernel_pid_t
    target_pid
```

```
int msg_try_receive(msg_t *m)
```



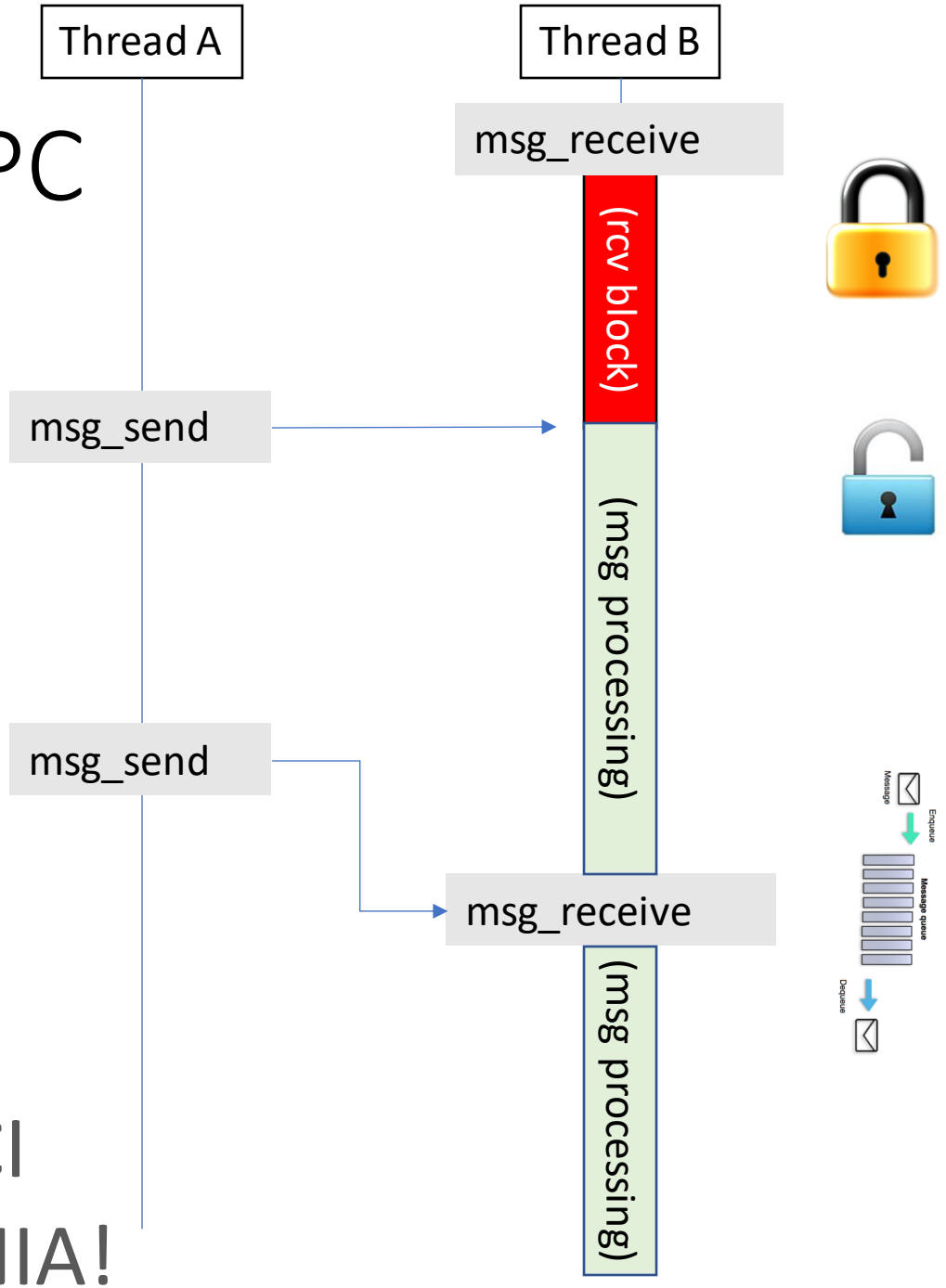
Kolejka wiadomości

- Wątek ma wydzieloną część pamięci dla wiadomości przychodzących z innych wątków
- Wątki mogą dodawać nowe wiadomości do kolejek innych wątków
- Wątki mogą „blokować” oczekiwanie na nową wiadomość – jeśli używane jest blokujące API



Komunikacja asynchroniczna IPC

```
static msg_t rcv_queue[RCV_QUEUE_SIZE];

void * thread_B_function(void* arg){
    msg_t msg;

    (void)arg;
    msg_init_queue(rcv_queue, RCV_QUEUE_SIZE);
    while (1) {
        msg_receive(&msg);
        …
    }
    return NULL;
}
```



WYKORZYSTYWANE W WIĘKSZOŚCI WĄTKÓW OBSŁUGUJĄCYCH ZDARZENIA!

ZADANIE

Komunikacja między wątkami

→ **Zadanie na komputerze**

→ **Instrukcja PDF**

A view on RIOT's console

```
>
> ps
        pid | name                 | state    Q | pri | stack ( used) | location
          1 | idle                 | pending  Q |  15 |   256 (  220) | 0x20000598
          2 | main                 | running  Q |   7 |  1536 (  696) | 0x20000698
          3 | pktdump              | bl rx    _ |   6 |  1536 (  248) | 0x20003784
          4 | 6lo                  | bl rx    _ |   3 |  1024 (  296) | 0x20003d94
          5 | ipv6                 | bl rx    _ |   4 |  1024 (  260) | 0x20001970
          6 | ieee802154_control   | bl rx    _ |   4 |  1024 (  440) | 0x200049a4
          7 | udp                  | bl rx    _ |   5 |  1024 (  272) | 0x2000459c
          8 | coap                 | bl rx    _ |   6 |  1536 (  556) | 0x20001344
          9 | ezradio2             | bl rx    _ |   2 |  1024 (  552) | 0x20000cf0
            | SUM                  |            |     |  9984 ( 3540)
>
```

Thread's state

Priority

Total and used memory in stack

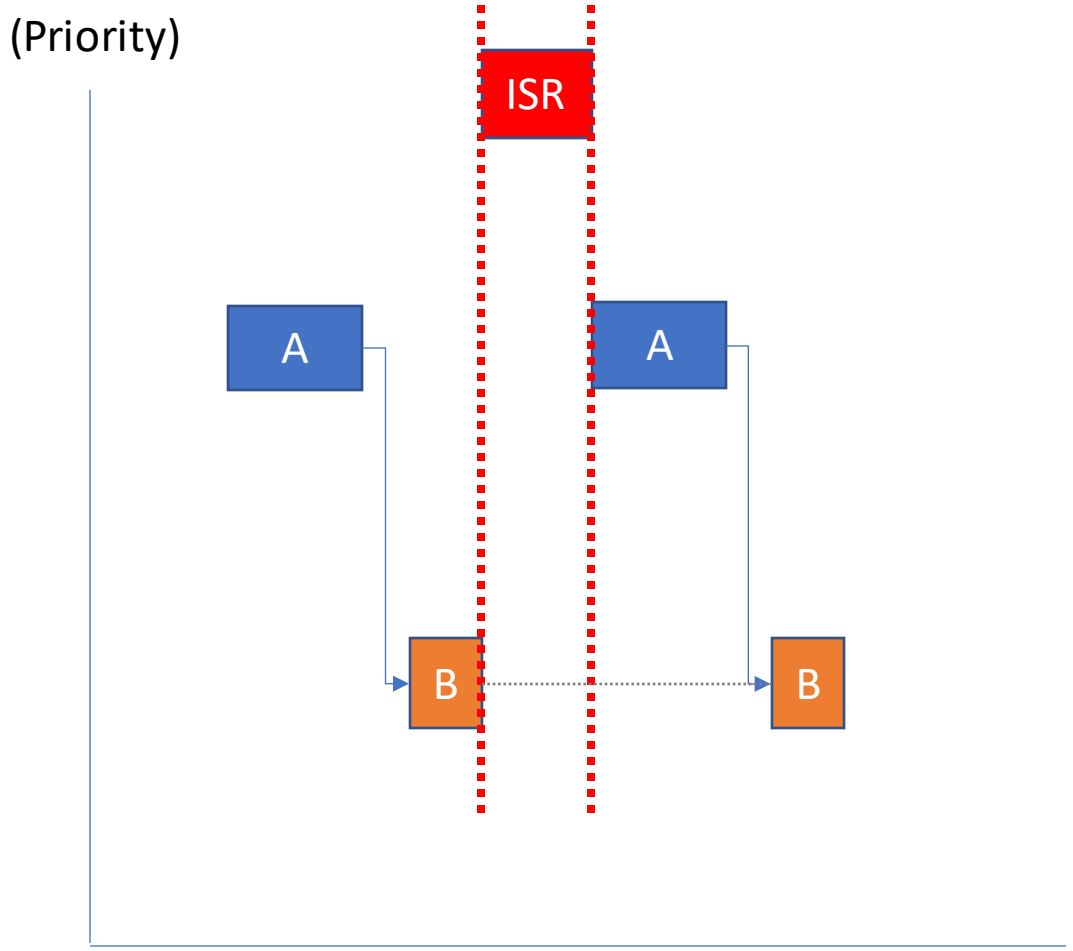
Stack start in memory

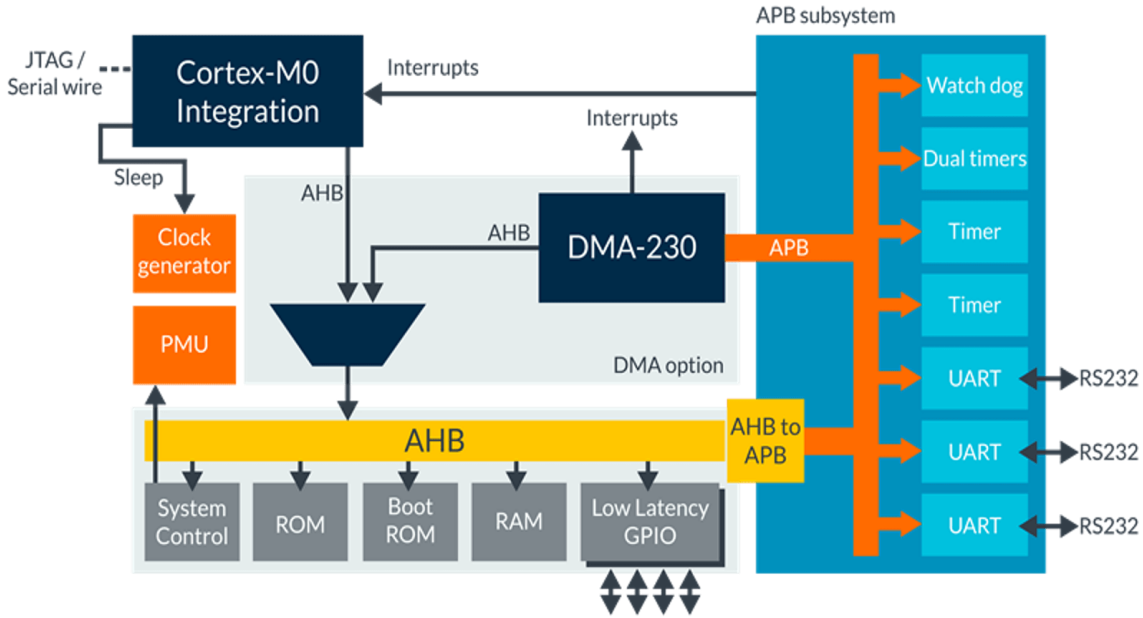
Nine (9) threads

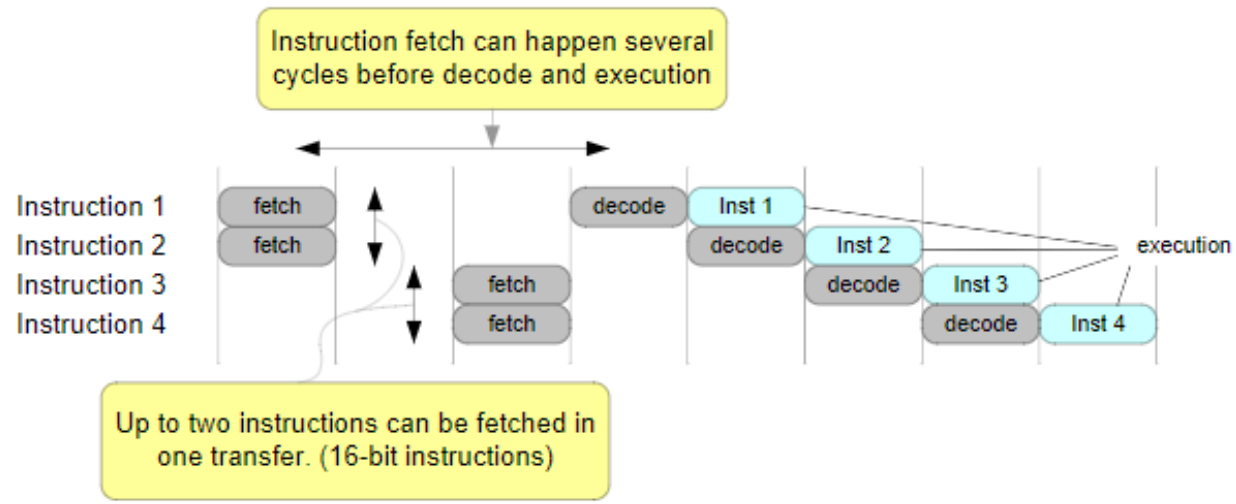
Concurrent memory access

When does it occur? Why does it occur? How can I prevent it?

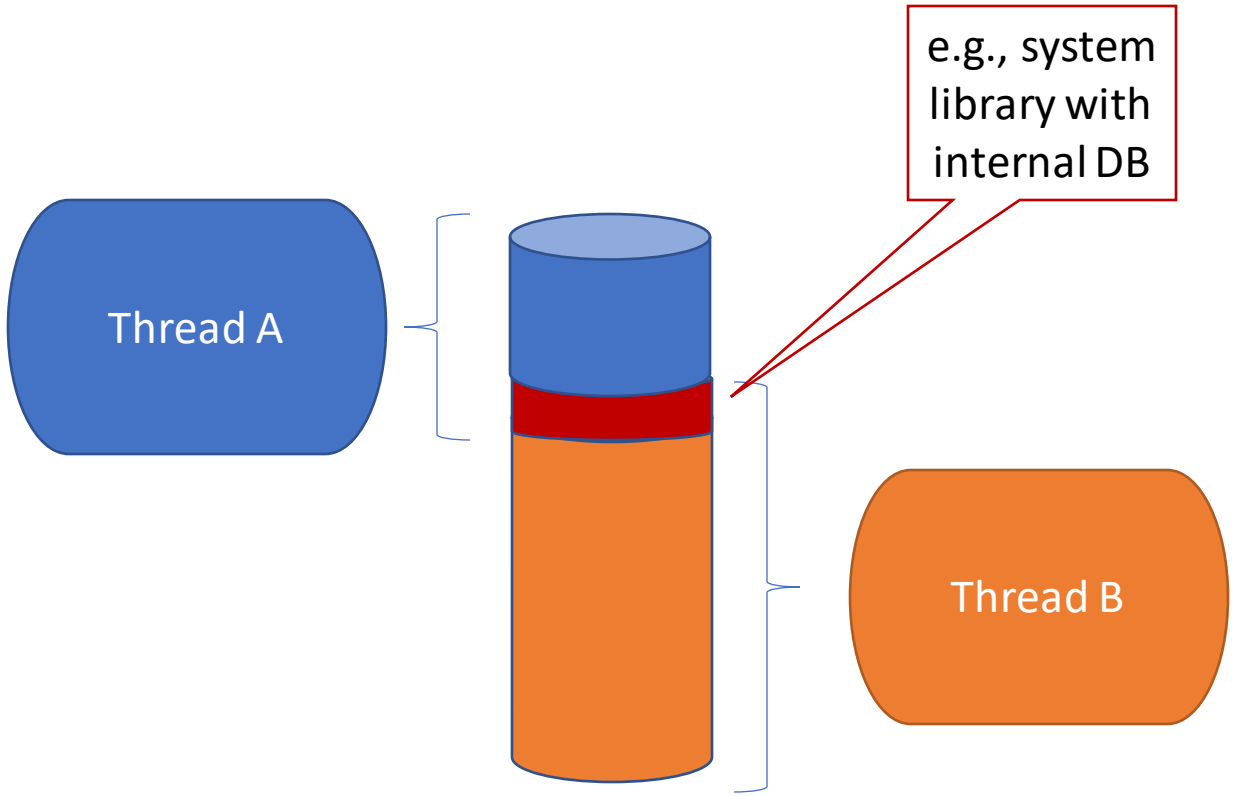
If there is only one CPU, is it possible to have concurrent memory access?




APB subsystem
JTAG / Serial wire
Cortex-M0 Integration
Interrupts
Interrupts
Sleep
AHB
AHB
DMA-230
APB
Clock generator
PMU
DMA option
AHB
AHB to APB
System Control
ROM
Boot ROM
RAM
Low Latency GPIO
Watch dog
Dual timers
Timer
Timer
UART
UART
UART
RS232
RS232
RS232


Instruction fetch can happen several cycles before decode and execution
Instruction 1
Instruction 2
Instruction 3
Instruction 4
fetch
fetch
fetch
fetch
decode
Inst 1
decode
Inst 2
decode
Inst 3
decode
Inst 4
execution
Up to two instructions can be fetched in one transfer. (16-bit instructions)

Thread-thread data access collision



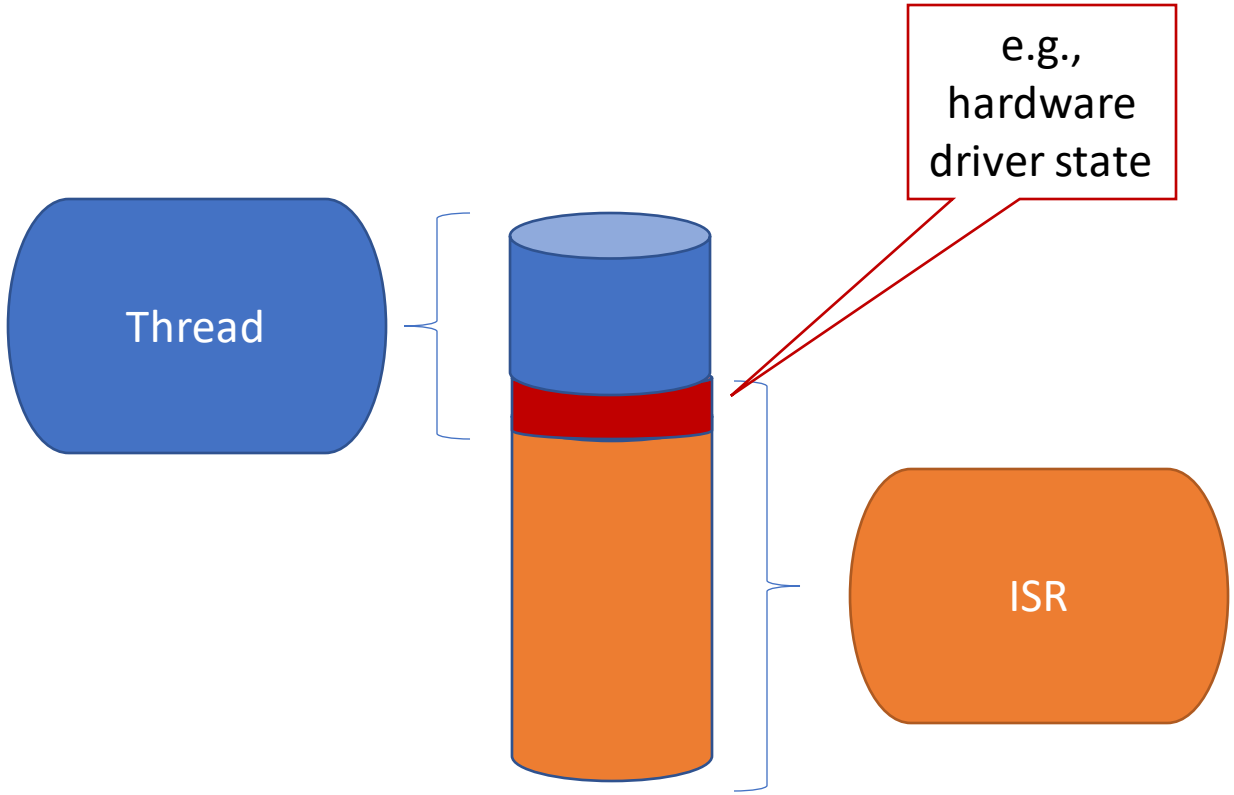
SOLUTION

```
mutex_t table_lock;                                    (global variable)
```

```
static void* _find_client(int id){
    void* result;
    mutex_lock(&table_lock);
    // Find in table
    mutex_unlock(&table_lock);
    return result;
}
```

```
static void _add_client(int id, void* details){
    mutex_lock(&table_lock);
    // Modify table
    mutex_unlock(&table_lock);
}
```

Data access collision involving (at least 1) ISR



SOLUTION

```
static void* _find_client(int id){
    void* result;
    unsigned int irq_state = irq_disable();
    // Find in table
    irq_restore(state);
    return result;
}
```

```
static void _add_client(int id, void* details){
    unsigned int irq_state = irq_disable();
    // Modify table
    irq_restore(state);
}
```

NO IRQs WILL BE PROCESSED DURING THIS TIME!!

ZADANIE

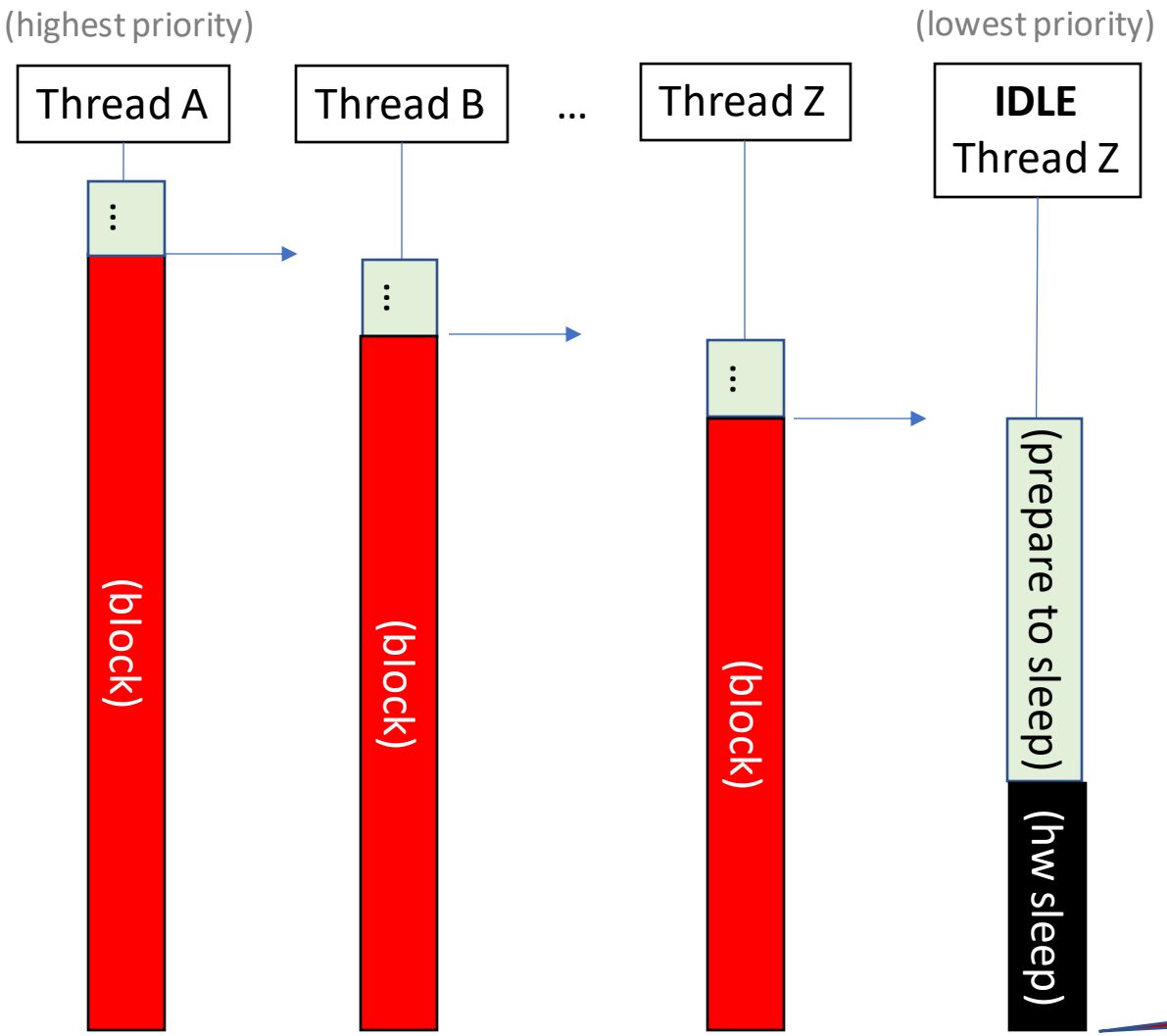
Dostęp do pamięci

→ Zadanie na komputerze

→ Instrukcja PDF

Energy savings

Condition for hw sleeping



Usually woken-up by a hardware interruption

A view on RIOT's console

```
>
> ps
    pid | name                 | state    Q | pri | stack ( used) | location
      1 | idle                 | pending  Q |  15 |   256 (  220) | 0x20000598
      2 | main                 | running  Q |   7 |  1536 (  696) | 0x20000698
      3 | pktdump              | bl rx    _ |   6 |  1536 (  248) | 0x20003784
      4 | 6lo                  | bl rx    _ |   3 |  1024 (  296) | 0x20003d94
      5 | ipv6                 | bl rx    _ |   4 |  1024 (  260) | 0x20001970
      6 | ieee802154_control   | bl rx    _ |   4 |  1024 (  440) | 0x200049a4
      7 | udp                  | bl rx    _ |   5 |  1024 (  272) | 0x2000459c
      8 | coap                 | bl rx    _ |   6 |  1536 (  556) | 0x20001344
      9 | ezradio2             | bl rx    _ |   2 |  1024 (  552) | 0x20000cf0
        | SUM                  |            |     |  9984 ( 3540)
>
```

Usypianie wątku

- RTOS zwykle używa tylko jednego uruchomionego timera sprzętowego (oszczędzanie energii)
- Zegar sprzętowy służy do emulacji kilku zegarów programowych
- Wątek może przejść w tryb uśpienia na określony czas

```
void send_frame_tsch(void *frame, uint32_t
timestamp){
        uint32_t gap = timestamp - xtimer_now();
        xtimer_usleep(gap);
        send_frame(frame);
}
```

PREZENTACJA

Dostęp do pamięci

→ Zadanie na komputerze

→ Instrukcja PDF