

# Publish-Subscribe

Programowanie systemowe i współbieżne

Kacper Lisiak 160241

v1.0, 2025-01-25

Projekt jest dostępny w repozytorium github:

<https://github.com/Kacper-spare/publish-subscribe>

## 1 Struktury danych

Projekt został wykonany z użyciem dwóch struktur danych.

1. struktura TQueue:

```
struct TQueue
{
    int capacity;
    int tail;
    void** messageArray;
    TNode* subscribers;

    pthread_cond_t lockGetMsg;
    pthread_cond_t lockAddMsg;
    pthread_cond_t lockEditing;
    pthread_mutex_t mutexGetMsg;
    pthread_mutex_t mutexAddMsg;
    pthread_mutex_t mutexEditing;

    int activeReaders;
};
```

wyróżniamy tutaj dwa rodzaje zmiennych

- zmienne synchronizujące

```
pthread_cond_t lockGetMsg;
pthread_cond_t lockAddMsg;
pthread_cond_t lockEditing;
pthread_mutex_t mutexGetMsg;
pthread_mutex_t mutexAddMsg;
pthread_mutex_t mutexEditing;
int activeReaders; //liczba wątków aktualnie czytających informacje kolejki
```

zmienne te służą wyłącznie do synchronizacji wątków w programie z umożliwieniem istnienia wielu różnych struktur TQueue

- zmienne kolejki TQueue

```
int capacity; //maksymalna ilość wiadomości w kolejce
int tail; //indeks ostatniej wiadomości w kolejce
void** messageArray; //lista dynamiczna zawierająca wiadomości
TNode* subscribers; //lista jednokierunkowa z id wątków zasubskrybowanych
```

zmienne te obsługują informacje kolejki odpowiednio opisane w komentarzu obok zmiennej. Zmienna subscribers jest stworzona przy użyciu struktury, której implementacja wygląda następująco.

## 2. Struktura TNode

```
typedef struct Node
{
    int head; //informacja o indeksie kolejnej wiadomości do przeczytania
    pthread_t* data; //wskaźnik do id wątku
    struct Node* next; //wskaźnik do kolejnego elementu listy
} TNode;
```

struktura ta umożliwia dynamiczną alokację i dealokację dodatkowych subskrybentów bez używania funkcji realloc() która może zwrócić NULL i usunąć całą listę. Sposób implementacji subskrybentów jest preferencją.

## 2 Funkcje

Poniższe funkcje zawarte tutaj są funkcjami związanymi z kolejką jednokierunkową pozostałe funkcje są zaimplementowane zgodnie z specyfikacją projektu.

1. TNode\* newNode(pthread\_t\* thread, TQueue\* queue) – tworzy nowy wskaźnik do kolejnego elementu listy.
2. TNode\* removeNode(TNode\* head) – usuwa element podany, funkcja ta jest zawsze używana dla pewnego podzbioru listy i zawsze usuwa pierwszy element podzbioru.

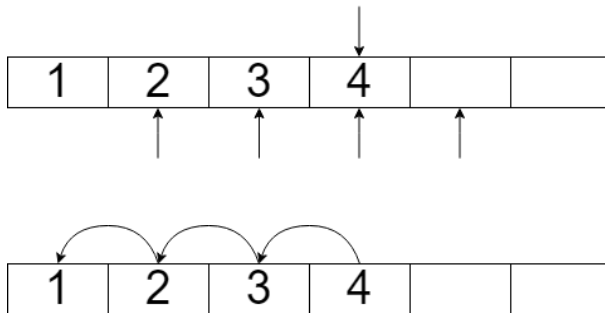
## 3 Algorytm

### 3.1 Implementacja

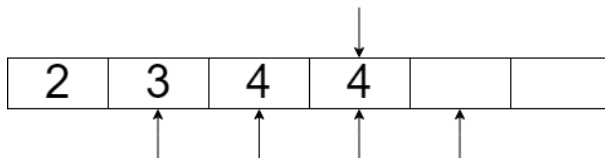
- 3.1.1 destroyQueue(TQueue\* queue) – niszczy kolejkę, musi zwolnić mutex związany z blokowaniem dostępu do kolejki, aby go usunąć, może więc powodować błędy typu read access violation.
- 3.1.2 setSize(TQueue\* queue, int size), unsubscribe(TQueue \*queue, pthread\_t thread), removeMsg(TQueue \*queue, void \*msg) – funkcje te korzystają z przesunięcia tablicy jednowymiarowej o n pozycji w lewo przykład:

górną strzałkę reprezentuje informację `tail` w strukturze `TQueue` dolne strzałki reprezentują informację `head` w zmiennej `subscribers`

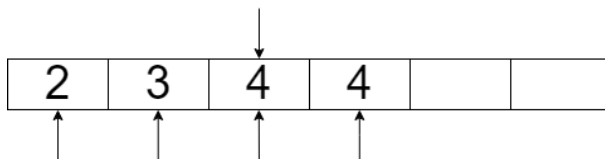
wykonamy przesunięcie o  $n = 1$



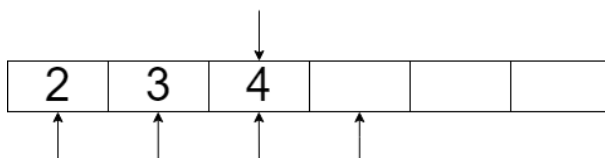
Na końcu przemieszczenia o  $n = 1$  otrzymujemy



Teraz algorytm poprawi wskaźniki przesuując je odpowiednio



I ostatecznie usunie zduplikowaną wiadomość (nie jest to jednak konieczne z perspektywy pozostałych funkcji)



**3.1.3** `unsubscribe(TQueue *queue, pthread_t thread)` – po usunięciu subskrybenta sprawdza czy jest jakaś wiadomość do usunięcia

## 3.2 Zabezpieczenia

W programie występuje dwa zabezpieczeń związanych z obsługą informacji w strukturze `TQueue`.

### 3.2.1 Sekcja krytyczna dla wątku czytającego

```
pthread_mutex_lock(&queue->mutexEditing);
queue->activeReaders++;
pthread_mutex_unlock(&queue->mutexEditing);
```

```

...

pthread_mutex_lock(&queue->mutexEditing);
queue->activeReaders--;
if (queue->activeReaders == 0)
{
    pthread_cond_broadcast(&queue->lockEditing);
}
pthread_mutex_unlock(&queue->mutexEditing);

```

Zabezpieczenie to informuje wątki edytujące informacje o istnieniu wątku, który aktualnie czyta informacje z TQueue, czyli informacja o tym, że nie powinien zmieniać tych informacji, dopóki inne wątki czytają je. Natomiast druga część informuje wątki uśpione przez zmienną warunkową o potencjalnej możliwości edycji TQueue. Informujemy wszystkie wątki, ponieważ kiedy jeden z nich skończy zadanie to drugi może rozpocząć swoje bez czekania aż znowu wystąpi sygnał na danej zmiennej warunkowej.

### 3.2.2 Sekcja krytyczna dla wątku edytującego

```

pthread_mutex_lock(&queue->mutexEditing);
while (queue->activeReaders != 0)
{
    pthread_cond_wait(&queue->lockEditing, &queue->mutexEditing);
}

...

pthread_mutex_unlock(&queue->mutexEditing);

```

Sekcja krytyczna występuje na całości funkcji (ponieważ edytujemy) co zatrzymuje inne wątki od edycji i czytania informacji w kolejce. Można ten problem porównać do problemu pisarzy i czytelników z priorytetem dla czytających.

### 3.2.3 Omówienie odporności na problemy przetwarzania współbieżnego

#### 3.2.3.1 Zakleszczenie

Nie powinno wystąpić. Zasobem żądanym przez wątki jest dostęp do struktury TQueue, wątki rywalizują o dostęp do tej struktury, ale jest ona chroniona jednym mutexem. Nie jest to jednak wystarczającym warunkiem do powstania zakleszczenia. Zakleszczenie mogłoby wystąpić, jeśli przy usypianiu odpowiedni mutex nie zostałby zwolniony.

#### 3.2.3.2 Aktywne oczekiwanie

Program nie zawiera pętli z uśpieniem typu `sleep(n)`, ani pętli które wyłącznie sprawdzają swój warunek (np. `while (queue->activeReaders != 0)`). Występują natomiast uśpienie jak przedstawione w [sekcji krytycznej dla wątku edytującego](#) które pozwala na uśpienie wątku i oczekiwanie na otrzymanie sygnału który obudzi wątek z powrotem do działania.

### 3.2.3.3 Głodzenie

Jak powiedziane wcześniej program jest podobny do problemu piszących i czytających z priorytetem dla czytających co pozwala na głodzenie wątków piszących. Dodanie kolejnego mutexa który by kontrolował zezwolenie na czytanie spowodowało by odwrócenie priorytetu (głodzenie wątków czytających), jednakże specyfikacja projektu mówi o blokowaniu się wątków, które próbują odebrać wiadomości, kiedy nie ma nowej wiadomości do odebrania. Na tej podstawie można stwierdzić że głodzenie może wystąpić jedynie kiedy wątki będą wywoływać funkcje `getAvailable(TQueue *queue, pthread_t thread)`. Odwrócenie priorytetu doprowadzi do jednakowego problemu poprzez wywoływanie funkcji edytujących.

## 4 Przykład użycia

Program, który będzie analizowany

```
void* function(void* args)
{
    subscribe((TQueue*) args, pthread_self());
    for (int i = 0; i < 10; i++)
    {
        printf("%d\n", *(int*)getMsg((TQueue*) args, pthread_self()));
    }
    unsubscribe((TQueue*) args, pthread_self());
}

int main()
{
    int sizeOfQueue = 10;
    int array[20] = {0};
    TQueue* queue = createQueue(sizeOfQueue);

    pthread_t threads[7];

    for (int i = 0; i < 7; i++)
    {
        pthread_create(&threads[i], NULL, function, (void*) queue);
    }
    for (int i = 0; i < 20; i++)
    {
        array[i] = i;
        addMsg(queue, (void*)&array[i]);
    }

    for (int i = 0; i < 7; i++)
    {
        pthread_join(threads[i], NULL);
    }
    destroyQueue(queue);
}
```

Program wywołuje kolejno `createQueue(sizeOfQueue)` następnie tworzy 7 wątków każdy z nich subskrybuje do kolejki `queue` następnie 10 razy odbiera wiadomość poprzez `getMsg((TQueue*)`

`args, pthread_self())` i wyświetla ją w terminalu a na koniec wywoła `unsubscribe(TQueue* queue)` z kolejki `queue`. W trakcie kiedy funkcja `function(void* args)` się wykonuje wielowątkowo wątek główny dodaje 20 wiadomości. Na koniec programu kolejka jest usuwana.

Faza	T0	T1	T2	T3	T4	T5	T6	T7
1:		sub(Q)	sub(Q)	sub(Q)	sub(Q)	sub(Q)	sub(Q)	
2:	add(0)							
3:	add(1)							
4:	add(2)							
5:								sub(Q)
6:	add(3)							
7:	add(4)							
8:	add(5)							
9:	add(6)							
10:	add(7)							
11:		0←get(Q)	0←get(Q)	0←get(Q)	0←get(Q)	0←get(Q)	0←get(Q)	3←get(Q)
12:							rm(Q,0)	
13:		1←get(Q)	1←get(Q)	1←get(Q)	1←get(Q)	1←get(Q)	1←get(Q)	4←get(Q)
14:							rm(Q,1)	
15:		2←get(Q)	2←get(Q)	2←get(Q)	2←get(Q)	2←get(Q)	2←get(Q)	5←get(Q)
16:							rm(Q,2)	
17:	add(8)							
18:	add(9)							
19:	add(10)							
20:	add(11)							
21:	add(12)							
22:	add(13)							
23:	↓	3←get(Q)	3←get(Q)	3←get(Q)	3←get(Q)	3←get(Q)	3←get(Q)	6←get(Q)
24:	↓						rm(Q,3)	
25:	13							
26:		4←get(Q)	4←get(Q)	4←get(Q)	4←get(Q)	4←get(Q)	4←get(Q)	8←get(Q)
27:							rm(Q,4)	
28:		5←get(Q)	5←get(Q)	5←get(Q)	5←get(Q)	5←get(Q)	5←get(Q)	9←get(Q)
29:							rm(Q,5)	

Kontynuując ten przykład dostaniemy końcowo, że wątki wypisały liczby do 0 do 9, natomiast wątek T7 wypisał liczby od 3 do 12. Program ma możliwość zakończenia się oczekiwaniem na otrzymanie sygnału od zmiennej warunkowej.