

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 nie ma żadnego realnego powodu takiej implementacji ponad inną.

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.

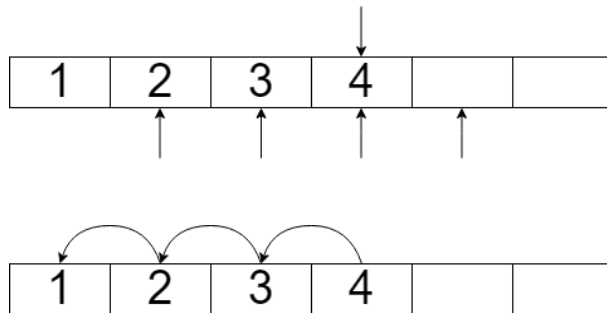
3 Algorytm

3.1 Implementacja

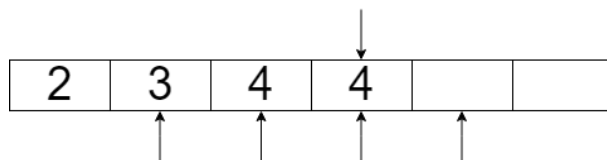
- 3.1.1 `destroyQueue(TQueue* queue)` – niszczy kolejkę, musi zwolnić mutex związany z blokowaniem dostępu do kolejki aby go usunąć, może więc powodować błędy typu `read access violation`.
- 3.1.2 `setSize(TQueue* queue, int size), unsubscribe(TQueue *queue, pthread_t thread), removeMsg(TQueue *queue, 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`

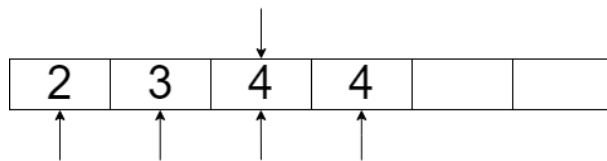
wykonamy przesunięcie o $n = 1$



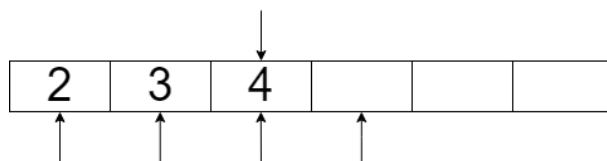
Na końcu przemieszczenia o $n = 1$ otrzymujemy



Teraz algorytm poprawi wskaźniki przesuwając je odpowiednio



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



3.2 Zabezpieczenia

W programie występuje kilka zabezpieczeń związanych z obsługą informacji kolejki.

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 wątko 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 czytających z priorytetem dla czytających.

3.2.3 Omówienie odporności na problemy przetwarzania współbieżnego

3.2.3.1 Zakleszczenie

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.