1 Treść zadania

Implementacja "bufora komunikacyjnego" - struktury danych mieszczącej maksymalnie `M` elementów, pozwalającej na wyjmowanie elementów w kolejności umieszczania i odwrotnej. Należy zadbać o zabezpieczenie przed czytaniem z pustego bufora, zapisem do pełnego bufora i jednoczesnym użyciem bufora przez różne procesy.

Bufor komunikacyjny o takiej samej funkcjonalności jak w poprzednim zadaniu, ma zostać zrealizowany za pomocą monitorów.

2 Założenia

1. Wielkość kolejki i segment pamięci współdzielonej są określane przy inicjalizacji programu
2. Typ elementów jest określony przed rozpoczęciem implementacji
3. Wykorzystywane są elementy kodu napisanego przy implementacji poprzedniego ćwiczenia.

3 Realizacja

Do zrealizowania zadania planuję stworzyć i wykorzystać następujące obiekty:

3.1 `Semaphore`

Obiekt semafora, będący interfejsem dostępu do funkcji systemowych semafora. Potrzebny do blokowania wejścia do monitora i czekania na warunek.

3.2 `Condition`

Obiekt warunku, zliczający oczekujących na jego spełnienie. Ma za zadanie zawieszać i odwieszać procesy w zależności od spełnienia warunku.

3.3 `Monitor`

Obiekt monitora, zabezpieczający przed jednoczesnym dostępem do tego samego zasobu i obsługujący zawieszanie i odwieszanie procesów w zależności od zadanego warunku (poprzez obiekt `Condition`)

3.4 `Queue`

Obiekt kolejki, obsługujący funkcje umieszczania i usuwania elementów w pamięci współdzielonej. Funkcje te są niezabezpieczone przed jednoczesnym dostępem.

3.5 `Buffer`

Obiekt bufora, obsługujący funkcje kolejki, ale zabezpieczający je przed jednoczesnym dostępem za pomocą monitora.

4 Planowana implementacja

Do zaimplementowania powyższych funkcji potrzebne będą następujące elementy:

4.1 `Semaphore`

Moduł `sem_func` z poprzedniego ćwiczenia zamieniony w klasę implementującą semafor. Metody:

- `proberen()` Jeśli semafor jest opuszczony zawiesza proces. Jeśli nie, przepuszcza proces i opuszcza semafor.
- `verhogen()` Podnosi semafor

4.2 `Condition`

Klasa umożliwiająca operowanie na warunkach.

Atrybuty:

- `waitCount` - licznik oczekujących procesów
- `internalSemaphore` - semafor kontrolujący spełnienie warunku

Metody:

- `wait()` Inkrementuje licznik i opuszcza semafor
- `signal()` Jeśli są oczekujący, dekrementuje licznik i podnosi semafor

4.3 `Monitor`

Klasa chroni przed jednoczesnym dostępem do swoich metod.
Atrybuty:

- `mutext` - obiekt klasy `Semaphore` zabezpieczający przed jednoczesnym dostępem

Metody:

- `enter()` Blokuje dostęp za pomocą semafora `mutex`
- `leave()` Odblokowuje dostęp za pomocą semafora `mutex`

4.4 Queue

Klasa implementująca funkcje modułu `Queue` z poprzedniego ćwiczenia, posiadające analogiczne atrybuty:

- `size` - rozmiar
- `id` - id pamięci współdzielonej
- `list` - miejsce przechowywania elementów

i metody:

- `push()`
- `popBack()`
- `popFront()`
- `print()` - zamiast `list()`
- `increment()`
- `decrement()`

Dodane zostaną jeszcze dwie metody:

- `getSize()` Zwracająca rozmiar kolejki (stały, podany przy uruchomieniu programu)
- `getCount()` Zwracająca ilość elementów aktualnie znajdujących się w kolejce

4.5 Buffer

Klasa bufora zapewniająca bezpieczny dostęp do kolejki: Atrybuty:

- `*notFull` - wskaźnik na obiekt klasy `Condition` określający warunek zapełnienia listy
- `*notEmpty` - wskaźnik na obiekt klasy `Condition` określający warunek pustości listy
- `*queue` - wskaźnik na obiekt klasy `Queue` z kolejką danego bufora

Metody będą analogiczne do modułu bufora z poprzedniego ćwiczenia:

- `push()`
- `popBack()`
- `popFront()`

W celu zapewnienia bezpiecznego dostępu do kolejki, klasa używa mechanizmu monitora do zablokowania użytkowania przez wiele procesów na raz. Dlatego klasa ta dziedziczy po klasie `Monitor`