SOI – Semafory

Koncepcja

1. Zadanie

Mamy bufor FIFO na liczby całkowite. * Procesy A1 generują kolejne liczby parzyste modulo 50, jeżeli w buforze jest mniej niż 10 liczb parzystych. * Procesy A2 generują kolejne liczby nieparzyste modulo 50, jeżeli liczb parzystych w buforze jest więcej niż nieparzystych. * Procesy B1 zjadają liczby parzyste pod warunkiem, że bufor zawiera co najmniej 3 liczby. * Procesy B2 zjadają liczby nieparzyste, pod warunkiem, że bufor zawiera co najmniej 7 liczb. W systemie może być dowolna liczba procesów każdego z typów. Zrealizuj wyżej wymienioną funkcjonalność przy pomocy semaforów. Zakładamy, że bufor FIFO poza standardowym put() i get() ma tylko metodę umożliwiającą sprawdzenie liczby na wyjściu (bez wyjmowania) oraz posiada metody zliczające elementy parzyste i nieparzyste. Zakładamy, że semafory mają tylko operacje P i V.

2. Schemat Ogólny

W zadaniu mamy cztery rodzaje procesów (A1, A2, B1, B2) działających na wspólnym buforze FIFO. Kluczowym jest użycie semaforów do synchronizacji tych procesów, z zachowaniem określonych warunków.

3. Struktury i Semafory

- Bufor FIFO:

    a. std::vector<int> buffer: Wektor do przechowywania danych.

    b. size_t capacity: Maksymalny rozmiar bufora.

    c. Semaphore mutex: Binarny semafor służący jako blokada do ochrony sekcji krytycznej (dostęp do bufora).

    d. Semaphore empty: Semafor do sygnalizowania, że bufor jest pusty. Jego początkowa wartość to capacity.

    e. Semaphore full: Semafor do sygnalizowania, że bufor jest pełny. Jego początkowa wartość to 0.

    f. Bufor ma również liczniki evenCount i oddCount do śledzenia liczby parzystych i nieparzystych elementów. Metody put i get używają semaforów full, empty, i mutex do koordynacji dostępu do bufora, zapewniając, że operacje są bezpieczne pod względem współbieżności.

4. Logika Procesów

- A1: Sprawdza, czy liczba parzystych jest mniejsza niż 10. Jeśli tak, dodaje kolejną liczbę parzystą modulo 50.

- A2: Sprawdza, czy liczb parzystych jest więcej niż nieparzystych. Jeśli tak, dodaje kolejną liczbę nieparzystą modulo 50.
- B1: "Zjada" liczbę parzystą, jeśli w buforze jest co najmniej 3 liczby.
- B2: "Zjada" liczbę nieparzystą, jeśli w buforze jest co najmniej 7 liczb.

5. Szkic implementacji

```
#include <semaphore>
#include <mutex>
#include <thread>

class FifoBuffer {
private:
   std::vector<int> buffer;
   size_t capacity;
   Semaphore mutex; // Do ochrony sekcji krytycznej
   Semaphore<> empty;  // Sygnalizuje, że bufor jest pusty
   Semaphore<> full;   // Sygnalizuje, że bufor jest pełny

public:
   FifoBuffer(size_t cap); // Konstruktor

   void put(int value); // Wstawia wartość do bufora
   int get(bool consumeEven); // Pobiera wartość z bufora
   int peek() const; // Podgląda pierwszy element w buforze
};

FifoBuffer buffer(bufferSize);

void processA2() {
   while (true) {
      if (buffer.countEven() > buffer.countOdd()) {
         buffer.put(generateOddNumber());
      }
   }
}

void processB1() {
   while (true) {

      if (buffer.count() >= 3 && isEven(buffer.peek())) {
         buffer.get();
      }
   }
}

void processB2() {
   while (true) {

      if (buffer.count() >= 7 && !isEven(buffer.peek())) {
         buffer.get();
```

```
    }

  }
}
int main() {
  // Kod do uruchomienia procesów
}
```

6. Wstępna implementacja FIFOBuffer

```
class FifoBuffer {
private:
    std::vector<int> buffer;
    size_t capacity;
    Semaphore mutex; // Semafor binarny do ochrony sekcji krytycznej
    Semaphore empty;  // Semafor do sygnalizowania, że bufor jest pusty
    Semaphore full;   // Semafor do sygnalizowania, że bufor jest pełny

    int evenCount;
    int oddCount;

public:
    FifoBuffer(size_t cap) : capacity(cap), mutex(1), empty(cap),
  full(0), evenCount(0), oddCount(0) {}

    void put(int value) {
        full.p(); // Czeka, aż będzie miejsce w buforze
        mutex.p(); // Blokuje dostęp do bufora
        buffer.push_back(value);
        updateCountsOnPut(value);
        mutex.v(); // Zwalnia dostęp do bufora
        empty.v(); // Sygnalizuje, że bufor nie jest pusty
    }

    int get(bool consumeEven) {
        int value = 0;
        empty.p(); // Czeka, aż bufor będzie miał elementy
        mutex.p(); // Blokuje dostęp do bufora
        if (!buffer.empty() && (consumeEven == isEven(buffer.front())))
  {
            value = buffer.front();
            buffer.erase(buffer.begin());
            updateCountsOnGet(value);
        }
        mutex.v(); // Zwalnia dostęp do bufora
        full.v(); // Sygnalizuje, że w buforze jest miejsce
        return value;
```

```
42.     }
43.
44.     int peek() const {
45.         if (!buffer.empty()) {
46.             return buffer.front();
47.         }
48.         return -1; // Lub inna wartość wskazująca na pusty bufor
49.     }
50.
51.     int countEven() const { return evenCount; }
52.     int countOdd() const { return oddCount; }
53.
54. private:
55.     void updateCountsOnPut(int value) {
56.         if (isEven(value)) {
57.             ++evenCount;
58.         } else {
59.             ++oddCount;
60.         }
61.     }
62.
63.     void updateCountsOnGet(int value) {
64.         if (isEven(value)) {
65.             --evenCount;
66.         } else {
67.             --oddCount;
68.         }
69.     }
70.
71.     bool isEven(int value) const {
72.         return value % 2 == 0;
73.     }
74. };
```

7. testowanie

Poprzez printowanie zawartości bufora i obserwowanie czy bufor zachowuje się jak należy