# SOI LAB 4 – Bufor komunikacyjny z użyciem monitorów

## Implementacja bufora

```
IngredientBuffer::IngredientBuffer() : full(), empty(), Monitor() {
    this->top = 0;
}

void IngredientBuffer::put_ingredient(Data ingredient) {
    this->enter();
    if (this->top == BUFFER_SIZE)
        this->wait( & full);

    this->data[this->top++] = ingredient;

    if (this->top == 1)
        this->signal( & empty);
    this->leave();
}

Data IngredientBuffer::get_ingredient() {
    Data ingredient;
    this->enter();
    if (this->top == 0)
        this->wait( & empty);

    ingredient = this->data[--this->top];

    if (this->top == BUFFER_SIZE - 1)
        this->signal( & full);
    this->leave();
    return ingredient;
}
```

Bufor który zaimplementowałem jest obiektem dziedziczącym po klasie bazowej monitora. Zawiera tablicę danych typu *Data*, przechowującą komunikaty (składniki pierogów) w formie stosu. Pole top jest szczytem stosu. Stała *BUFFER_SIZE* określa maksymalny rozmiar bufora. W kodzie ustawiona jest na 10. Obiekty tej klasy posiadają także pola *full* oraz *empty*, będące zmiennymi warunkowymi pozwalającymi na zapewnienie ochrony przed czytaniem z pustego bufora oraz wkładaniu elementów na pełny bufor.

W przypadku wkładania elementów funkcją *put_ingredient* na początku blokujemy monitor metodą *enter*(), a następnie sprawdzamy czy bufor jest pełny. Jeśli jest, to oczekujemy na sygnalizację zmiennej warunkowej *full*, w przeciwnym wypadku dodajemy element do bufora. Jeśli jest to pierwszy element na buforze, oznacza to że przestał być pusty, więc sygnalizowana jest zmienna *empty*.

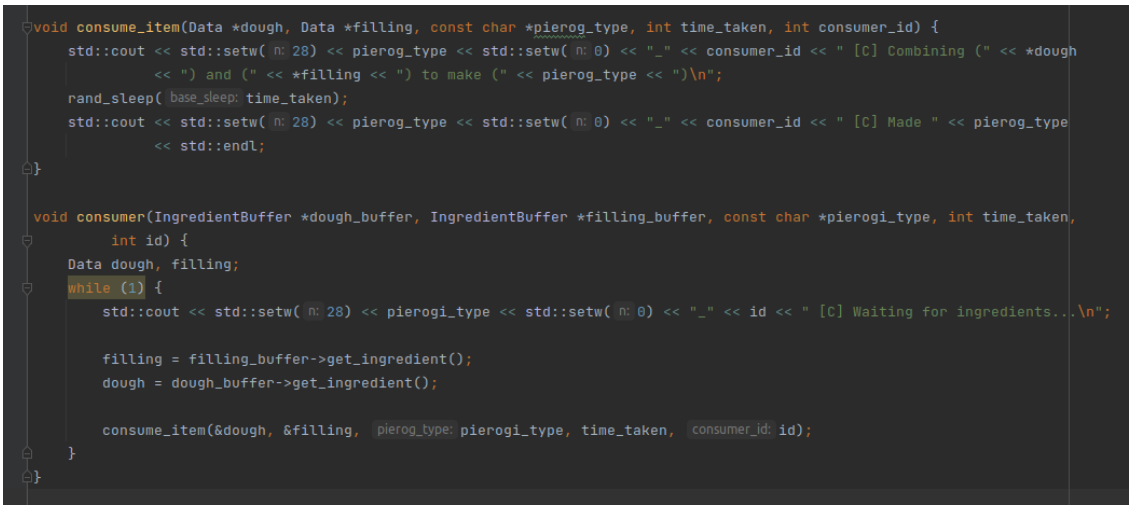
Przy czytaniu z bufora także trzeba go zablokować metodą *enter*(), po czym sprawdzane jest czy bufor jest pusty. Jeśli tak, to oczekujemy na zmienną warunkową *empty*, w przeciwnym wypadku wyjmujemy element z bufora. Jeśli był to element ze szczytu bufora, to należy zasygnalizować zmienną *full*, gdyż zwolniło się miejsce.

# Implementacje klas zmiennych warunkowych, monitora oraz semafora

W kodzie został użyty kod dostępny pod adresem:

*https://www.ia.pw.edu.pl/~tkruk/edu/soib/lab/monitor.h*

# Implementacja producenta i konsumenta

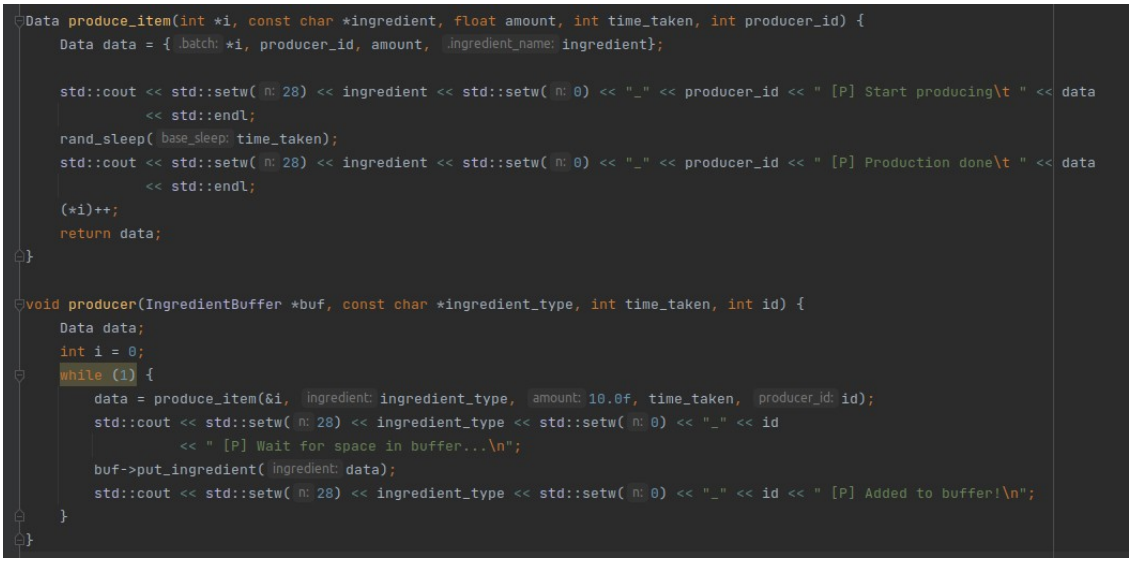


```
void consume_item(Data *dough, Data *filling, const char *pierog_type, int time_taken, int consumer_id) {
    std::cout << std::setw( n: 28) << pierog_type << std::setw( n: 0) << "_" << consumer_id << " [C] Combining (" << *dough
              << ") and (" << *filling << ") to make (" << pierog_type << ")\n";
    rand_sleep( base_sleep: time_taken);
    std::cout << std::setw( n: 28) << pierog_type << std::setw( n: 0) << "_" << consumer_id << " [C] Made " << pierog_type
              << std::endl;
}

void consumer(IngredientBuffer *dough_buffer, IngredientBuffer *filling_buffer, const char *pierogi_type, int time_taken,
        int id) {
    Data dough, filling;
    while (1) {
        std::cout << std::setw( n: 28) << pierogi_type << std::setw( n: 0) << "_" << id << " [C] Waiting for ingredients...\n";

        filling = filling_buffer->get_ingredient();
        dough = dough_buffer->get_ingredient();

        consume_item(&dough, &filling,  pierog_type: pierogi_type, time_taken,  consumer_id: id);
    }
}
```



```
Data produce_item(int *i, const char *ingredient, float amount, int time_taken, int producer_id) {
    Data data = { batch: *i, producer_id, amount,  ingredient_name: ingredient};

    std::cout << std::setw( n: 28) << ingredient << std::setw( n: 0) << "_" << producer_id << " [P] Start producing\t " << data
              << std::endl;
    rand_sleep( base_sleep: time_taken);
    std::cout << std::setw( n: 28) << ingredient << std::setw( n: 0) << "_" << producer_id << " [P] Production done\t " << data
              << std::endl;
    (*i)++;
    return data;
}

void producer(IngredientBuffer *buf, const char *ingredient_type, int time_taken, int id) {
    Data data;
    int i = 0;
    while (1) {
        data = produce_item(&i,  ingredient: ingredient_type,  amount: 10.0f, time_taken,  producer_id: id);
        std::cout << std::setw( n: 28) << ingredient_type << std::setw( n: 0) << "_" << id
                  << " [P] Wait for space in buffer...\n";
        buf->put_ingredient( ingredient: data);
        std::cout << std::setw( n: 28) << ingredient_type << std::setw( n: 0) << "_" << id << " [P] Added to buffer!\n";
    }
}
```

W przypadku konsumentów przyjmowanymi argumentami są 2 bufory – jeden od ciasta a drugi od składnika który jest potrzebny do ulepienia pieroga danego typu. Konsumenci także przyjmują argumenty czasu konsumpcji oraz ich ID. Dzięki użyciu monitorów, konsument jak i producent nie muszą się przejmować tworzeniem sekcji krytycznej. Monitor zapewnia to swoją strukturą. Dzięki temu wystarczy jedynie pobrać obydwa składniki metodami *get_ingridient*() i skonsumować obydwa (ulepić pieróg). Producenci natomiast przyjmują jako argument jeden bufor do którego odkładają wyprodukowane przez siebie składniki danego typu (*ingredient_type*), swoje ID unikalne w grupie producentów zajmujących się tym samym składnikiem oraz to ile czasu zajmuje produkcja. Na początku produkowany jest składnik danego typu, o danej masie i danym numerze. W tym wypadku sytuacja także jest prosta dzięki monitorowi. Wystarczy przywołać metodę *put_ingridient*.

# Program testowy

Program można skompilować przy użyciu polecenia cmake:

```
$cmake CmakeLists.txt
$make
```

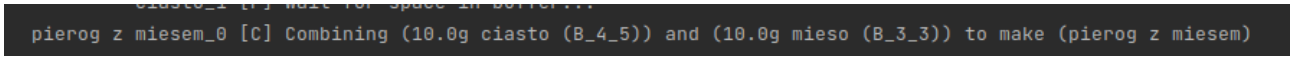
Uruchomienie programu:

```
$./lab4 <argumenty wywołania>
Gdzie argumentami wywołania to po kolei:
```

- *rand_fraction → losowość czasu oczekiwania [0;1]*
- *dough_prod_time → czas potrzebny na produkcję ciasta [ms]*
- *meat_prod_time → czas potrzebny na produkcję mięsa [ms]*
- *cheese_prod_time → czas potrzebny na produkcję twarogu [ms]*
- *cabb_prod_time → czas potrzebny na produkcję kapusty [ms]*
- *meat_cons_time → czas potrzebny na lepienie pieroga z mięsem [ms]*
- *cheese_cons_time → czas potrzebny na lepienie pieroga z twarogiem [ms]*
- *cabb_cons_time → czas potrzebny na lepienie pieroga z kapustą [ms]*
- *n_dough_prod → ilość producentów ciasta*
- *n_meat_prod → ilość producentów mięsa*
- *n_cheese_prod → ilość producentów twarogu*
- *n_cabb_prod → ilość producentów kapusty*
- *n_meat_cons → ilość konsumentów lepiących pierogi z mięsem*
- *n_cheese_cons → ilość konsumentów lepiących pierogi z twarogiem*
- *n_cabb_cons → ilość konsumentów lepiących pierogi z kapustą*

W wypadku pominięcia któregoś z argumentów użyte będą ich podstawowe wartości zdefiniowane w kodzie źródłowym.

W funkcji *main* znajduje się alokacja buforów dla ciasta, mięsa, sera oraz kapusty na pamięci współdzielonej. Następnie są one inicjalizowane. Po wykonaniu tych czynności wywoływane są funkcje *create_n_producers* oraz *create_n_consumers*. Jako argumenty przyjmują one bufory których dotyczy dana grupa konsumentów/producentów, to jakim produktem się zajmują, ile takich konsumentów/producentów ma być oraz wektor w którym składowane są wszystkie PID stworzonych procesów. Każdy z konsumentów/producentów wypisuje na standardowe wyjście informacje o tym czym się w danym momencie zajmuje wedle poniższego schematu:




Od lewej do prawej są to po kolei:

- produkt którym zajmuje się producent/konsument

- po znaku ‘_’ - numer producenta/konsumenta danego typu

- w kwadratowym nawiasie litera C przypomina, że jest to konsument a litera P, że jest to producent

- cała reszta wiersza jest opisem akcji która właśnie się zaczęła/została ukończona

W tym wypadku jest to komunikat konsumenta który właśnie zaczął tworzyć pieróg z mięsem (zaczął konsumpcję, ma id=0).