

Kamil Breczko

Problem producenta i konsumenta

11 stycznia 2017

## Spis treści

|   |   |
|---|---|
| <b>1. Opis zadania</b>                  | 3 |
| 1.1. Problem producenta i konsumenta    | 3 |
| 1.2. Założenia                          | 3 |
| 1.3. Przykłady                          | 3 |
| 1.4. Sposób rozwiązania                 | 4 |
| <b>2. Opis programu</b>                 | 5 |
| 2.1. Pamięć dzielona                    | 5 |
| 2.2. Stan początkowy                    | 5 |
| 2.3. Produkowanie i pobieranie elementu | 5 |
| 2.4. Proces producenta                  | 6 |
| 2.5. Proces konsumenta                  | 6 |
| <b>3. Uruchamianie</b>                  | 7 |
| 3.1. Dane szczegółowe                   | 7 |
| 3.2. Sposób uruchamiania                | 7 |
| 3.3. Przykład                           | 7 |
| <b>4. Testowanie</b>                    | 8 |

## 1. Opis zadania

### 1.1. Problem producenta i konsumenta

Problem producenta i konsumenta to klasyczny informatyczny problem synchronizacji. Dotyczy przekazywania danych – w szczególności strumienia danych – pomiędzy procesami z wykorzystaniem bufora o ograniczonej pojemności. Występują dwa rodzaje procesów: producent i konsument. Producenci produkują pewne elementy i umieszczają je w buforze o ograniczonym rozmiarze. Konsumenti pobierają elementy z bufora i je konsumują. Z punktu widzenia producenta problem synchronizacji polega na tym, że nie może on umieścić kolejnej jednostki, jeśli bufor jest pełny. Z punktu widzenia konsumenta problem synchronizacji polega na tym, że nie powinien on mieć dostępu do bufora, jeśli nie ma tam żadnego elementu do pobrania.

### 1.2. Założenia

Założenia do rozwiązania problemu producenta i konsumenta:

- producenci produkują pewne elementy i umieszczają je w buforze;
- konsumenci pobierają elementy z bufora i je konsumują;
- producenci i konsumenci przejawiają swą aktywność w nie dających się określić momentach czasu;
- bufor ma ograniczoną pojemność;
- gdy są wolne miejsca w buforze, producent może tam umieścić swój element;
- gdy w buforze brak miejsca na elementy, producent musi czekać, aż wolne miejsca się pojawią;
- gdy w buforze są jakieś elementy, konsument je pobiera;
- gdy brak elementów w buforze, konsument musi czekać, aż jakiś element się pojawi;
- dane nie są zmieniane, zanim nie zostaną odczytane;
- dane są odczytywane tylko raz;
- dane są odczytywane w kolejności, w której zostały wyprodukowane;
- wszystkie dane są w końcu odczytane;
- początkowo magazyn jest pusty;

### 1.3. Przykłady

Poniższa tabela przedstawia, w jakich sytuacjach możemy spotkać problem konsumenta i producenta.

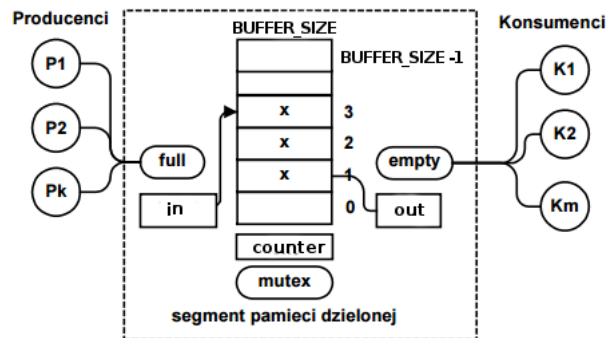
Tabela 1.1. Przykłady zależności producent-konsument

| Producent                | Konsument                |
|--------------------------|--------------------------|
| Klawiatura               | System operacyjny        |
| Edytor tekstu            | Drukarka                 |
| Joystick                 | Gra komputerowa          |
| Gra komputerowa          | Ekran                    |
| Łącza komunikacyjne      | Przeglądarka internetowa |
| Przeglądarka internetowa | Łącza komunikacyjne      |
| Aplikacja klienta        | Baza danych              |

#### 1.4. Sposób rozwiązania

Wykorzystanie dwóch semaforów ogólnych: *full* – będzie interpretowany jako liczba elementów bufora, które można skosztować i *empty* – będzie interpretowany jako liczba wolnych elementów w buforze. Dodatkowo użycie semafora nienazwanego *mutex* będzie zapewniało wzajemne wykluczanie dostępu do zmiennych dzielonych.

Aby skontrolować działanie programu, producent wstawia kolejne liczby naturalne do bufora, a konsument pobiera wyprodukowane przez producenta liczby.



Rysunek 1.1. Sposób rozwiązania problemu

## 2. Opis programu

### 2.1. Pamięć dzielona

W celu komunikacji między procesami, wszystkie utworzone procesy konsumenta i producenta, dzielą tą samą pamięć:

```
struct Data{
    int counter;
    int buffer[BUFFER_SIZE];
    sem_t empty;
    sem_t full;
    sem_t mutex;
    int in;
    int out;
};
```

Za pomocą funkcji *mmap*, gdzie *shared\_data* wskazuje na strukturę *Data*. Odwzorowuje plik w pamięci danego procesu. Funkcja zwraca wskaźnik do właśnie odwzorowanej pamięci.

```
shared_data = mmap(0, sizeof(Data), PROT_READ|PROT_WRITE, MAP_SHARED| MAP_ANONYMOUS, -1, 0);
```

### 2.2. Stan początkowy

Przy założeniu, że początkowo bufor jest pusty, wartość semafora *empty* wynosi *BUFFER\_SIZE*, a semafora *full* wynosi 0 (żadna pozycja w buforze nie jest zajęta).

Wskaźnik *in* wskazuje następne miejsce wstawienia nowego elementu, a wskaźnik *out* wskazuje następne miejsce do pobrania w buforze. Oba mają wartość początkową równą 0.

W programie użyty został także licznik *counter*, który będzie interpretowany jako liczba elementów w buforze. Początkowo przyjmuje wartość 0 zgodnie z założeniem w p. 1.2. Stan początkowy wspólnych zmiennych procesów producenta i konsumenta przedstawia się następująco:

```
buffer[BUFFER_SIZE]={0,0,...,0};
in = 0;
out = 0;
counter = 0;
full = 0;
empty = BUFFER_SIZE;
mutex = 1;
```

### 2.3. Produkowanie i pobieranie elementu

Producent, wstawiając element, używa dzielonej zmiennej *in*, która wskazuje kolejną pozycję do zapełnienia w buforze. Zmienna jest zwiększana cyklicznie (modulo *BUFFER\_SIZE*) po każdym wstawieniu elementu do bufora. Jednostki nowo produkowane są przekazywane przez argument funkcji *put*, która zwraca element wcześniejszy, czyli 0.

```
int put(int value){
    int tmp= shared_data->buffer[shared_data->in];
    shared_data->buffer[shared_data->in] = value;
    shared_data->in = (shared_data->in + 1) % BUFFER_SIZE;
    return tmp;
}
```

Analogicznie konsument, pobierając element, używa dzielonej zmiennej *out*, która wskazuje kolejną pozycję do pobrania z buforu. Zmienna jest zwiększana cyklicznie (modulo *BUFFER\_SIZE*) po każdym odebraniu elementu z buforu. Funkcja *get* powinna zwrócić liczbę różną od 0.

```
int get(){
    int tmp = shared_data->buffer[shared_data->out];
    shared_data->buffer[shared_data->out] = 0;
    shared_data->out = (shared_data->out + 1) % BUFFER_SIZE;
    return tmp;
}
```

## 2.4. Proces producenta

Wstawienie elementu do bufora jest poprzedzone operacją opuszczenia semafora *empty*, gdzie wartość 0 oznacza brak wolnego miejsca w buforze. W ten sposób producent blokowany jest w dostępie do bufora, co chroni bufor przed przepełnieniem. Semafor *empty* zostanie podniesiony przez konsumenta, gdy zwolni on miejsce w buforze. Następnie, jeśli producentowi uda się umieścić kolejny element, sygnalizuje to przez podniesienie semafora *full*, dając znak konsumentowi, że dostępny jest nowy element do odebrania i zwiększa liczbę elementów w buforze o 1. Proces producenta jest zdefiniowany następująco:

```
void producer() {
    int i=1;
    while (1) {
        sem_wait(&(shared_data->empty));
        sem_wait(&(shared_data->mutex));
        -----Sekcja krytyczna-----
        shared_data->counter++;
        i++;
        printf("[counter=%d]Produced! [ %d ] <====%d\n", shared_data->counter, put(i),i);
        -----        Koniec        -----
        sem_post(&(shared_data->mutex));
        sem_post(&(shared_data->full));
        sleep(2);
    }
}
```

## 2.5. Proces konsumenta

Proces konsumenta jest symetryczny. Przed uzyskaniem dostępu do bufora, konsument wykonuje operację opuszczenia semafora *full*, który jest zwiększany przez producenta po umieszczeniu w buforze kolejnego elementu. Jeśli semafor *full* jest równy 0, bufor jest pusty i konsument utknie w operacji opuszczania. W przypadku gdy konsument uzyska dostęp do bufora, pobierze element i tym samym zwolni miejsce poprzez zmniejszenie liczby elementów w buforze o 1. Fakt ten zasignalizuje poprzez podniesienie semafora *empty*, co z kolei umożliwi wykonanie kolejnego kroku producentowi. Proces konsumenta jest zdefiniowany następująco:

```
void consumer() {
    while (1) {
        sem_wait(&(shared_data->full));
        sem_wait(&(shared_data->mutex));
        -----Sekcja krytyczna-----
        shared_data->counter--;
        printf("[counter=%d]Consumed! [ %d ] ==> \n", shared_data->counter, get());
        -----        Koniec        -----
        sem_post(&(shared_data->mutex));
        sem_post(&(shared_data->empty));
        sleep(2);
    }
}
```

### 3. Uruchamianie

#### 3.1. Dane szczegółowe

Program został wykonany zgodnie z informacjami zamieszczonymi w tabeli 2.1.

Tabela 2.1. Wymagania do uruchomienia programu

|                     |                  |
|---------------------|------------------|
| Język Programowania | C                |
| Kompilator          | GNU GCC Compiler |
| System operacyjny   | Linux            |

#### 3.2. Sposób uruchamiania

Aby uruchomić program, należy skompilować plik poleceniem:

```
gcc -o ProducerAndConsumer main.c -pthread
```

lub za pomocą przygotowanego pliku makefile:

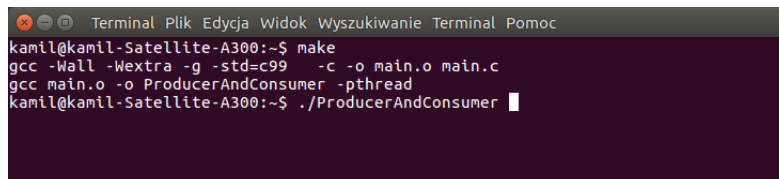
```
make
```

Następnie, w obu przypadkach, wpisać w terminal:

```
./ProducerAndConsumer
```

#### 3.3. Przykład

Na rysunku 3.1 został przedstawiony przykład poprawnego uruchomionego programu.

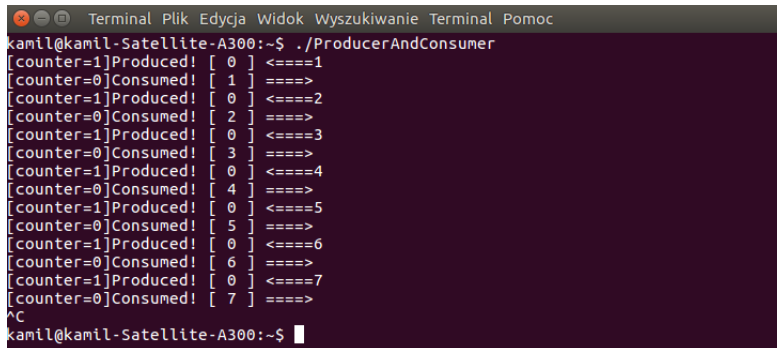


```
Terminal Plik Edycja Widok Wyszukiwanie Terminal Pomoc
kamil@kamil-Satellite-A300:~$ make
gcc -Wall -Wextra -g -std=c99 -c -o main.o main.c
gcc main.o -o ProducerAndConsumer -pthread
kamil@kamil-Satellite-A300:~$ ./ProducerAndConsumer
```

Rysunek 3.1. Poprawnie uruchomiony program

## 4. Testowanie

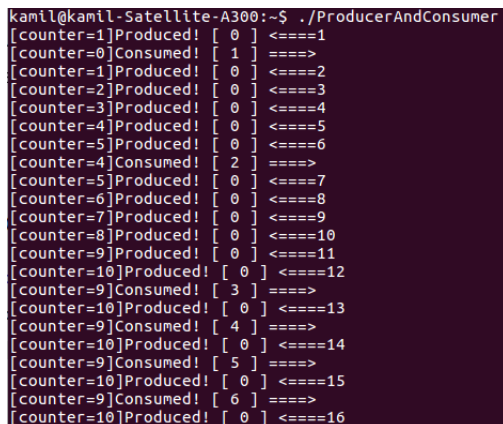
Semafor *mutex* zapewnia wzajemne wykluczanie wewnątrz sekcji krytycznej (daje wyłączny dostęp albo producentowi, albo konsumentowi). Para semaforów *empty* oraz *full* realizuje kontrolę zapelnienia bufora. Tym samym liczba elementów w buforze (licznik *counter*) nie będzie ujemna ani większa od maksymalnej pojemności bufora.



```
kamil@kamil-Satellite-A300:~$ ./ProducerAndConsumer
[counter=1]Produced! [ 0 ] <====1
[counter=0]Consumed! [ 1 ] <====>
[counter=1]Produced! [ 0 ] <====2
[counter=0]Consumed! [ 2 ] <====>
[counter=1]Produced! [ 0 ] <====3
[counter=0]Consumed! [ 3 ] <====>
[counter=1]Produced! [ 0 ] <====4
[counter=0]Consumed! [ 4 ] <====>
[counter=1]Produced! [ 0 ] <====5
[counter=0]Consumed! [ 5 ] <====>
[counter=1]Produced! [ 0 ] <====6
[counter=0]Consumed! [ 6 ] <====>
[counter=1]Produced! [ 0 ] <====7
[counter=0]Consumed! [ 7 ] <====>
^C
kamil@kamil-Satellite-A300:~$
```

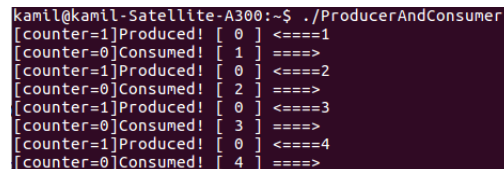
Rysunek 4.1. Testowanie Programu

Przy uruchomieniu programu możemy zauważyć, że wszystkie wcześniej wymienione założenia zostały spełnione. Początkowo bufor jest pusty, więc program zaczyna od umieszczenia elementu. Producent umieszcza nowe elementy tak, że dane nie są zastępowane, zanim nie zostaną odczytane. Widzimy, że zmienianą wartością zawsze jest 0. Konsument odczytuje zawsze element tylko raz i to w kolejności jak zostały wyprodukowane, tzn. nigdy nie odbierze wartości 0.



```
kamil@kamil-Satellite-A300:~$ ./ProducerAndConsumer
[counter=1]Produced! [ 0 ] <====1
[counter=0]Consumed! [ 1 ] <====>
[counter=1]Produced! [ 0 ] <====2
[counter=2]Produced! [ 0 ] <====3
[counter=3]Produced! [ 0 ] <====4
[counter=4]Produced! [ 0 ] <====5
[counter=5]Produced! [ 0 ] <====6
[counter=4]Consumed! [ 2 ] <====>
[counter=5]Produced! [ 0 ] <====7
[counter=6]Produced! [ 0 ] <====8
[counter=7]Produced! [ 0 ] <====9
[counter=8]Produced! [ 0 ] <====10
[counter=9]Produced! [ 0 ] <====11
[counter=10]Produced! [ 0 ] <====12
[counter=9]Consumed! [ 3 ] <====>
[counter=10]Produced! [ 0 ] <====13
[counter=9]Consumed! [ 4 ] <====>
[counter=10]Produced! [ 0 ] <====14
[counter=9]Consumed! [ 5 ] <====>
[counter=10]Produced! [ 0 ] <====15
[counter=9]Consumed! [ 6 ] <====>
[counter=10]Produced! [ 0 ] <====16
```

(a) Konsument jest usypiany na dłuższy czas.



```
kamil@kamil-Satellite-A300:~$ ./ProducerAndConsumer
[counter=1]Produced! [ 0 ] <====1
[counter=0]Consumed! [ 1 ] <====>
[counter=1]Produced! [ 0 ] <====2
[counter=0]Consumed! [ 2 ] <====>
[counter=1]Produced! [ 0 ] <====3
[counter=0]Consumed! [ 3 ] <====>
[counter=1]Produced! [ 0 ] <====4
[counter=0]Consumed! [ 4 ] <====>
```

(b) Producent jest usypiany na dłuższy czas.

Rysunek 4.2. Kontrola bufora

Nawet jeżeli uruchomimy kilku producentów i konsumentów, to będą oni poprawnie korzystać ze wspólnego bufora. Kolejność, w jakiej budzone są procesy oczekujące na podniesienie semafora gwarantuje, że nie wystąpi zagłodzenie.