

Aleksandra Sikora

Systemy operacyjne – problem producenta i konsumenta

14 grudnia 2016

Spis treści

1. Opis zadania.....	3
1.1. Problem producenta i konsumenta.....	3
1.2. Sposób rozwiązania.....	3
2. Opis programu.....	4
2.1. Proces producenta	5
2.2. Proces konsumenta.....	6
3. Urachamianie i testowanie.....	7

1. Opis zadania

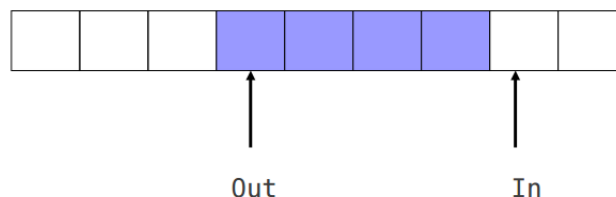
1.1. Problem producenta i konsumenta

Problem producenta i konsumenta to klasyczny informatyczny problem synchronizacji. Na problem składają się proces producenta i konsumenta, którzy dzielą zasób – bufor dla produkowanych (konsumowanych) jednostek. Zadaniem producenta jest wytworzenie produktu, umieszczenie go w buforze i rozpoczęcie pracy od nowa. W tym samym czasie konsument ma pobrać produkt z bufora. Problem polega na takiej synchronizacji, aby producent nie dodawał nowych jednostek, gdy bufor jest pełny, a konsument nie pobierał, gdy bufor jest pusty. Rozwagać można dwie wersje tego problemu – z ograniczonym buforem lub, nieco uproszczoną, z nieograniczonym buforem.

1.2. Sposób rozwiązania

Rozwiązaniem dla producenta jest uspienie procesu w momencie, gdy bufor jest pełny. Pierwszy konsument, który pobierze element z bufora budzi proces producenta, który uzupełnia bufor. W analogiczny sposób usypiany jest konsument, który próbuje pobrać z pustego bufora. Pierwszy producent, po dodaniu nowego produktu, umożliwi dalsze działanie konsumentowi.

W rozwiązaniu dołączonym do dokumentacji wykorzystuję komunikację międzyprocesową z użyciem semaforów oraz rozważam wersję problemu z ograniczonym buforem dla wielu producentów i konsumentów. Stosuję bufor cykliczny z dwoma wskaźnikami *in* i *out*. Proces producenta zapisuje wyprodukowaną jednostkę pod adresem, który określa wskaźnik *in*, natomiast proces konsument pobiera jednostkę określaną przez wskaźnik *out*.



2. Opis programu

W rozwiązaniu dołączonym do dokumentacji używam trzech semaforów: *empty*, *full*, *mutex*. Semafor *empty* jest opuszczany przed dodaniem jednostki do bufora przez producenta. Gdy bufor jest pełny, semafor nie może być opuszczony i proces producenta zostaje zatrzymany. Analogicznie semafor *full* zostaje opuszczony przed pobraniem jednostki z bufora przez konsumenta. Tak samo jak w przypadku producenta, przed opuszczeniem semafora zostaje sprawdzony warunek, czy bufor nie jest pusty. Semafor *mutex* rozwiązuje problem sekcji krytycznej, zapewniając wzajemne wykluczanie przy dostępie do zmiennych dzielonych, którymi są wskaźniki *in*, *out* wskazujące odpowiednio na pierwsze puste miejsce w buforze oraz na pierwsze zajęte miejsce w buforze.

```
typedef struct
{
    int buf[BUFF_SIZE];
    int in;
    int out;
    sem_t full;
    sem_t empty;
    pthread_mutex_t mutex;
} sbuf_t;

sbuf_t shared;
```

2.1. Proces *producenta*

Producent tworzy jednostkę, następnie opuszcza semafor *empty* mówiący o liczbie wolnych miejsc w buforze. Stara się również opuścić semafor *mutex*. Jeśli nie będzie to możliwe, czyli jego wartość będzie mniejsza niż 0, proces producenta zostanie wstrzymany. W przeciwnym przypadku producent przesunie wskaźnik *in* na następną komórkę bufora, po czym podniesie zarówno semafor *empty*, jak i *mutex*.

```
void *Producer(void *arg)
{
    int i, item, index;
    index = (int)arg;
    for (i=0; i < NITEMS; i++)
    {
        item = i;
        sem_wait(&shared.empty);
        pthread_mutex_lock(&shared.mutex);
        shared.buf[shared.in] = item;
        shared.in = (shared.in+1)%BUFF_SIZE;
        printf("[P%d] Producing %d ...\n", index, item);
        fflush(stdout);
        pthread_mutex_unlock(&shared.mutex);
        sem_post(&shared.full);
        if (i % 2 == 1) sleep(1);
    }
    return NULL;
}
```

Wykorzystanie bufora cyklicznego:

```
shared.buf[shared.in] = item;
```

Po wykonaniu swojego zadania proces producenta zostaje uśpiony na jedną milisekundę. Dzięki temu będzie spełniony warunek ograniczonego czekania.

```
if (i % 2 == 1) sleep(1);
```

2.2. Proces *konsumenta*

Konsument w pierwszej kolejności opuszcza semafor *full* mówiący o liczbie zajętych miejsc w buforze. Następnie podobnie jak proces producenta, stara się opuścić semafor *mutex*. Jeśli nie będzie to możliwe, proces konsumenta zostaje wstrzymany. W przeciwnym przypadku producent przesunie wskaźnik *out* na następną komórkę bufora, po czym podniesie zarówno semafor *full*, jak i *mutex*.

```
void *Consumer(void *arg)
{
    int i, item, index;
    index = (int)arg;
    for (i=NITEMS; i > 0; i--) {
        sem_wait(&shared.full);
        pthread_mutex_lock(&shared.mutex);
        item=i;
        item=shared.buf[shared.out];
        shared.out = (shared.out+1)%BUFF_SIZE;
        printf("[C%d] Consuming  %d ...\n", index, item);
        fflush(stdout);
        pthread_mutex_unlock(&shared.mutex);
        sem_post(&shared.empty);
        if (i % 2 == 1) sleep(1);
    }
    return NULL;
}
```

Wykorzystanie bufora cyklicznego w przypadku procesu konsumenta:

```
shared.out = (shared.out+1)%BUFF_SIZE;
```

Proces konsumenta, analogicznie do procesu producenta, po wykonaniu swojego zadania również zostaje uśpiony na jedną milisekundę.

```
if (i % 2 == 1) sleep(1);
```

3. Uruchamianie i testowanie

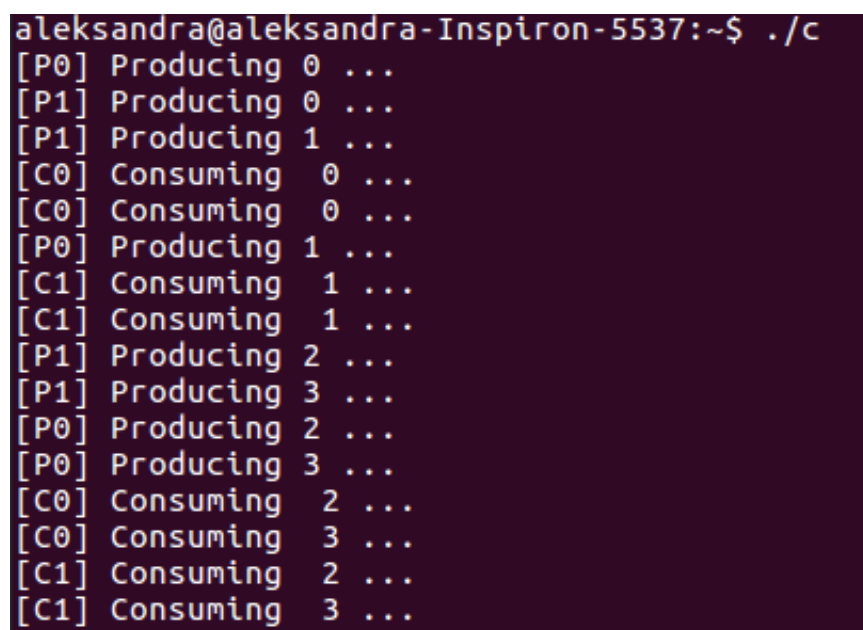
Do uruchomienia programu wymagany jest system Linux. W celu skompilowania programu należy użyć polecenia:

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

Następnie uruchamiany program poleceniem:

```
./c
```

Zdjęcie poniżej pokazuje przykładowe działanie:



```
aleksandra@aleksandra-Inspiron-5537:~$ ./c
[P0] Producing 0 ...
[P1] Producing 0 ...
[P1] Producing 1 ...
[C0] Consuming 0 ...
[C0] Consuming 0 ...
[P0] Producing 1 ...
[C1] Consuming 1 ...
[C1] Consuming 1 ...
[P1] Producing 2 ...
[P1] Producing 3 ...
[P0] Producing 2 ...
[P0] Producing 3 ...
[C0] Consuming 2 ...
[C0] Consuming 3 ...
[C1] Consuming 2 ...
[C1] Consuming 3 ...
```