SYSTEMY OPERACYJNE

Problem producenta i konsumenta
SITO LICZB

Karol Rodziński

3 stycznia 2018

Contents

1. Treść zadania

Napisz w języku C z użyciem semaforów program synchronizujący procesy . Objaśnij cel (założenia) danej synchronizacji oraz udokumentuj zarówno program, jak i sposób jego testowania. Jako problem do rozwiązania możesz obrać któryś z kilkunastu klasycznych problemów synchronizacji, skorzystać z licznych zadań tego typu prezentowanych w Sieci lub obmyślić własne (wyżej punktowane). W Sieci są dziesiątki problemów tego typu i ich gotowych rozwiązań. Nie chodzi o to, żeby je skopiować. Chodzi o to, żeby zrozumieć.

2. Kilka słów o tym problemie

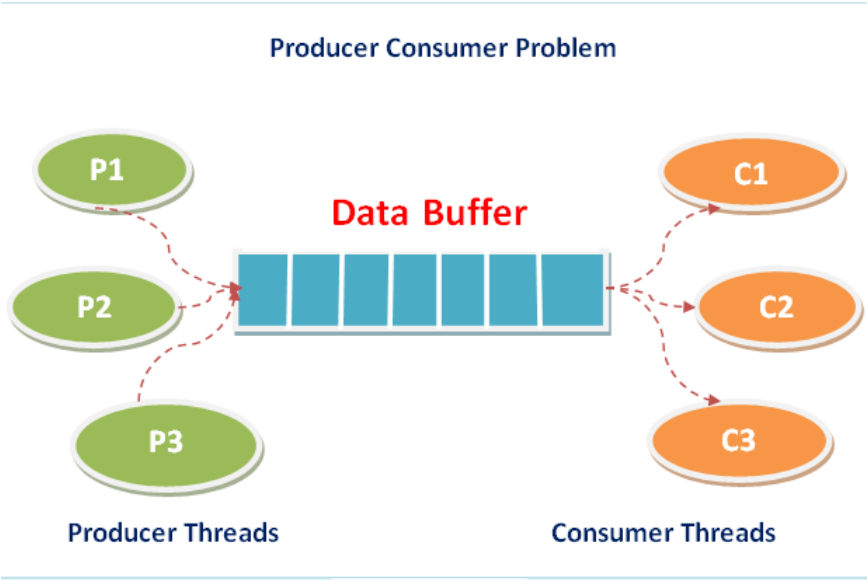
2.1 Ogólne założenia

Zagadnienie producenta-konsumenta (ang. producer-consumer problem) stanowi model współpracy procesów. Zakładamy, że proces producent wytwarza dane, które konsumuje proces konsumenta. Aby było to możliwe musi istnieć bufor (wyobraźmy sobie go jako tablicę) jednostek, który będzie zapełniany przez producenta i opróżniany przez konsumenta. W takim układzie, gdy producent produkuje jednostkę, konsument może wyciągnąć z bufora inną.

Ważne jest jednak, aby procesy producenta i konsumenta podlegały synchronizacji tak, by konsument nie "wyciągał" jednostek jeszcze niewyprodukowanych, a w przypadku pustego bufora poczekał na wyprodukowanie czegoś do "wyciągania".

Problem producenta-konsumenta może przyjmować różne postacie:

- z nieograniczonym buforem (ang. unbounded-buffer) - możliwa jest sytuacja oczekiwania przez konsumenta na nowe jednostki, jednakże producent produkuje je nieustannie,
- z ograniczonym buforem (ang. bounded-buffer) - możliwa jest sytuacja oczekiwania przez konsumenta na nowe jednostki, ale możliwe jest również, że czekał będzie producent w wyniku zapełnienia bufora.



Zdjęcie 1 Iustracja tego problemu (wątki zamiast procesów)

2.2 Pseudokod rozwiązania tego problemu

```
mutex buffer_mutex;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

def producer() {
  while (true) {
    item = produceItem();
    down(emptyCount);
    down(buffer_mutex);
    putItemIntoBuffer(item);
    up(buffer_mutex);
    up(fillCount);
  }
}

def consumer() {
  while (true) {
    down(fillCount);
    down(buffer_mutex);
    item = removeItemFromBuffer();
    up(buffer_mutex);
    up(emptyCount);
    consumeItem(item);
  }
}
```

Za angielską wikipedią:

"Since we want to use program with many producers and consumers and we do not want to have a problem with race condition - we need a way to execute a critical section with mutual exclusion. Notice that the order in which different semaphores are incremented or decremented is essential: changing the order might result in a deadlock.

It is important to note here that though mutex seems to work as a semaphore with value of 1 (binary semaphore), but there is difference in the fact that mutex has ownership concept. Ownership means that mutex can only be "incremented" back (set to 1) by the same process that "decremented" it (set to 0), and all others tasks wait until mutex is available for decrement (effectively meaning that resource is available), which ensures mutual exclusivity and avoids deadlock. Thus using mutexes improperly can stall many processes when exclusive access is not required, but mutex is used instead of semaphore."

2.3 "Sito liczb" - czyli moja wersja tego problemu

Pragnę przedstawić pewną wariację na temat tego problemu. Mianowicie w moim przypadku producent wstawia do wspólnego bufora danych liczby - począwszy od 0. Po czym zostaje uśpiony na czas 0 do 3 sekund (jest on generowany przez fukncję rand() i dzielony modulo 4). W tym czasie dwa procesy oczekują na dane z bufora i działają w następujący sposób:

- consumer_odd: sprawdza czy liczba w buforze jest nieparzysta; jeśli tak, to pobiera i wypisuje odpowiedni komunikat. Po takiej operacji proces zostaje uśpiony na czas od 0 do 9 sekund (jest on generowany przez fukncję rand() i dzielony modulo 10)
- consumer_even: sprawdza czy liczba w buforze jest parzysta; jeśli tak, to pobiera i wypisuje odpowedni komunikat. Po takiej operacji proces zostaje uśpiony na czas od 0 do 9 sekund (jest on generowany przez fukncję rand() i dzielony modulo 10)

Wartość początkowa bufora tj. size_of_buffer jest równa 0 i przy każdym dodaniu elementu, zwiększamy ją o 1, a w przypadku pobrania elementu przez proces konsumeta - zmniejszamy o 1. W sytuacji gdy bufor zostanie napełniony do pewnej stałej tj. MAX_SIZE_OF_BUFFER (definiuję ją używając dyrektywy #define - w programie dołączonym jest do niej przydziolna wartość 10), wówczas producent czeka, aż któryś z procesów opróżni bufor czy to z parzystego, czy też nieparzystego elementu w nim zawartego.

3. Opis rozwiązania

3.1 Producent

W metodzie, która uruchoma logikę producenta definijemy zmienną value_to_add, która jest umieszczana przez niego w buforze (kolejne liczby naturalne). Producent opuszcza semfafor empty i następnie wstawia ten element do bufora. Brak wolnego miejsca oznacza wartość 0 semafora empty i tym samym uniemożliwia opuszczenie. W ten sposób producent blokowany jest w dostępie do bufora, gdyby chciał dodać większą ilość elementów niż dopuszczona. Semafor empty zmieni swój status przez konsumenta, gdy pobierze on element z bufora. Jeśli producentowi uda się dodać kolejny element, to podnosi on semafor empty, tym samym dając znak konsumentowi, że został umieszczony nowy element w buforze.

```
void producer() {
  int value_to_add = 0;
  while (1) {
    sem_wait(&(shared_buffer->empty));
    sem_wait(&(shared_buffer->mutex));

    shared_buffer->size_of_buffer++;
    value_to_add++;
    put_element_to_shared_buffer(value_to_add);
    printf(" PID: %d, I have just produced (%d), current size: %d \n", getpid(), value_to_add, shared_buffer->size_of_buffer);

    sem_post(&(shared_buffer->mutex));
    sem_post(&(shared_buffer->full));
    sleep(rand() % 4);
  }
}
```

Zdjęcie 2 Metoda przedstawiająca logikę producenta

3.2 Konsument

Konsument wykonuje operację opuszczenia semafora full, który zwiększa producent po umieszczeniu w buforze kolejnego elementu. W przypadku gdy konsument uzyska dostęp do bufora, wówczas sprawdzi on parzystość elementu i jeśli jest ona odpowiednia dla niego (parzysta - even i nieparzysta - odd), wówczas pobierze element i tym samym zwolni miejsce poprzez zmniejszenie zmiennej size_of_buffer, a także podniesie on semafor empty, co oznacza, że producent może dodać kolejny element to bufora. Jeśli jednak ta weryfikacja się nie powiedzie, to podniesie semafor full, żeby drugi konsument mógł spróbować uzyskać tą liczbę i zostanie uśpiony. Różnicą w logice konsumenta nieparzystego i parzystego jest operator porównania.

```
void consumer_even() {
  while (1) {
    sem_wait(&(shared_buffer->full));
    sem_wait(&(shared_buffer->mutex));

    int value_to_take = get_element_from_shared_buffer();

    if (value_to_take % 2 == 0) {
      shared_buffer->size_of_buffer--;
      printf(" PID: %d, I have just consumed even number (%d), current size: %d \n", getpid(), value_to_take, shared_buffer->size_of_buffer);
      sem_post(&(shared_buffer->mutex));
      sem_post(&(shared_buffer->empty));
    }
    else {
      sem_post(&(shared_buffer->mutex));
      sem_post(&(shared_buffer->full));
    }
    sleep(rand() % 10);
  }
}
```

Zdjęcie 3 Metoda przedstawiająca logikę konsumenta

3.3 Bufor dzielony przez procesy

- put_element_to_shared_buffer: Zmienna head (pozycja, na mamy wstawić element) zwiększana jest o wartość modulo MAX_BUFFER_SIZE, co powoduje, że możemy "zapętlić" nasz bufor.
- get_element_from_shared_buffer: Zmienna tail, (pozycja do pobrania z buforu) jest zwiększana cyklicznie modulo MAX_BUFFER_SIZE po każdym pobraniu elementu.

```
int put_element_to_shared_buffer(int value) {
  int valueToRetrun = shared_buffer->buffer[shared_buffer->head];
  shared_buffer->buffer[shared_buffer->head] = value;
  shared_buffer->head = (shared_buffer->head + 1)  % MAX_SIZE_OF_BUFFER;
  return valueToRetrun;
}

int get_element_from_shared_buffer() {
  int valueToRetrun = shared_buffer->buffer[shared_buffer->tail];
  shared_buffer->buffer[shared_buffer->tail] = 0;
  shared_buffer->tail = (shared_buffer->tail + 1)  % MAX_SIZE_OF_BUFFER;
  return valueToRetrun;
}
```

Zdjęcie 4 Metoda przedstawiająca metody dostępne dla bufora

Struktura bufora:

- int size_of_buffer - ilość elementów w buforze
- int buffer[MAX_SIZE_OF_BUFFER] - przechowywanie elementów
- sem_t empty - semafor informujący o tym czy bufor jest pusty
- sem_t full - semafor informujący o tym czy bufor jest pełny
- sem_t mutex - semafor zapobiegający zjawisku race condition
- int head - pozycja, na ktoraw wstawic element
- int tail - pozycja do pobrania z bufora

```
16
17  struct Data{
18    int size_of_buffer;
19    int buffer[MAX_SIZE_OF_BUFFER];
20    sem_t empty;
21    sem_t full;
22    sem_t mutex;
23    int head;
24    int tail;
25  } * shared_buffer;
26
```

Zdjęcie 5 Metoda przedstawiająca logikę konsumenta

4. Uruchamianie i testowanie

```
vagrant@precise64:/vagrant/SRC$ uname -a
Linux precise64 3.2.0-23-generic #36-Ubuntu SMP Tue Apr 10 20:39:51 UTC 2012 x86_64 x86_64 x86_64 GNU/Linux
```

Zdjęcie 6 Informacje o systemie

```
gcc -std=gnu99 -Wall -Wextra -pthread main.c -o run.o
./run.o
```

Bądź też za pomocą polecenia make:

```
make
./run.o
```

W celu pozbycia się pliku wynikowego run można użyć polecenia

```
make clean
```

Sprawdźmy, czy na prawdę używamy trzech procesów, stopując go na chwilę i wykonując polecenie:

ps

```
vagrant@precise64:/vagrant/SRC$ gcc -std=gnu99 -Wall -Wextra -pthread main.c -o run.o
vagrant@precise64:/vagrant/SRC$ ./run.o
 PID: 3500, I have just produced (1), current size: 1
 PID: 3501, I have just consumed Odd number (1), current size: 0
 PID: 3500, I have just produced (2), current size: 1
 PID: 3502, I have just consumed even number (2), current size: 0
 PID: 3500, I have just produced (3), current size: 1
 PID: 3501, I have just consumed Odd number (3), current size: 0
 PID: 3500, I have just produced (4), current size: 1
 PID: 3502, I have just consumed even number (4), current size: 0
 PID: 3500, I have just produced (5), current size: 1
 PID: 3501, I have just consumed Odd number (5), current size: 0
 PID: 3500, I have just produced (6), current size: 1
 PID: 3500, I have just produced (7), current size: 2
 PID: 3502, I have just consumed even number (6), current size: 1
 PID: 3501, I have just consumed Odd number (7), current size: 0
 PID: 3500, I have just produced (8), current size: 1
 PID: 3500, I have just produced (9), current size: 2
 PID: 3502, I have just consumed even number (8), current size: 1
 PID: 3501, I have just consumed Odd number (9), current size: 0
 PID: 3500, I have just produced (10), current size: 1
 PID: 3500, I have just produced (11), current size: 2
 PID: 3502, I have just consumed even number (10), current size: 1
 PID: 3500, I have just produced (12), current size: 2
 PID: 3501, I have just consumed Odd number (11), current size: 1
 PID: 3502, I have just consumed even number (12), current size: 0
 PID: 3500, I have just produced (13), current size: 1
 PID: 3500, I have just produced (14), current size: 2
 PID: 3500, I have just produced (15), current size: 3
 PID: 3500, I have just produced (16), current size: 4
 PID: 3501, I have just consumed Odd number (13), current size: 3

[1]+  Stopped                 ./run.o
vagrant@precise64:/vagrant/SRC$ ps
  PID TTY          TIME CMD
 2431 pts/0    00:00:00 bash
 3500 pts/0    00:00:00 run.o
 3501 pts/0    00:00:00 run.o
 3502 pts/0    00:00:00 run.o
```

Zdjęcie 7 Procesy uruchmione w trakcie wykonywania programu

Po dłuższej obserwacji działania programu - producent przestaje dodawać i czeka na pobranie elementu, gdy bufor jest pełny, konsumenci pobierają elementy przeznaczone dla nich. Patrząc także na powyższe zdjęcie widzimy trzy procesy (producent i dwóch konsumentów), zatem wnioskujemy, że program działa tak, jak tego chcieliśmy.