Podstawy programowania współbieżnego

III

PROBLEMY SYNCHRONIZACYJNE I NARZĘDZIA SYNCHRONIZACJI (kontynuacja II)

Problem producenta i konsumenta

Definicje

bufor – wspólne dla procesów miejsce w pamięci do przechowywania danych w postaci zbioru elementów (tablica, lista, itd.) na, którym są zdefiniowane operacje zapisu i odczytu danych. Zakłada się, że zapis lub odczyt danych odbywa się atomowo.

komunikacja synchroniczna – procesy przekazują sobie dane bez pośrednictwa bufora

komunikacja asynchroniczna – procesy przekazują sobie dane za pośrednictwem bufora

Konsument – proces, który wykonuje odczyt danych z bufora

Producent – proces, który wykonuje zapis danych do bufora

1. Producent-konsument – wariant z buforem nieskończonym

Założenia:

a) jest skończona liczba producentów i konsumentów, które działają w pętlach nieskończonych wykonując operacje odczytu lub zapisu na buforze

b) bufor przed pierwszym zapisem zapisem jest pusty (składa się z pustych elementów)

c) dane są zapisywane do kolejnych elementów bufora. Po wykonaniu operacji zapisu do elementu bufora staje się on pełny, a kolejny zapis będzie wykonywany do następnego elementu bufora.

d) dane są odczytywane z kolejnych elementów bufora. Po wykonaniu operacji odczytu z elementu bufora staje się on pusty, a kolejny odczyt będzie wykonywany z następnego elementu bufora.

Własność bezpieczeństwa

konsument nie wykona odczytu z pustego elementu bufora

Własność żywotności

konsument zawsze wykona odczyt danych z bufora

uwaga: procesy producenta zawsze mogą zapisywać dane do kolejnych elementów bufora bo bufor jest nieskończony

2. Producent-konsument wariant z buforem skończonym

Założenia:

a) jest skończona liczba producentów i konsumentów, które działają w pętlach nieskończonych wykonując operacje odpowiednio zapisu lub odczytu na buforze

b) bufor przed pierwszym zapisem jest pusty (składa się z pustych elementów)

c) producent zapisuje dane do kolejnych elementów bufora. Po zapisie do ostatniego elementu bufora, następny zapis będzie wykonywany do pierwszego elementu bufora.

d) konsument odczytuje dane z kolejnych elementów bufora. Wykonaniu operacji odczytu elementu bufora powoduje, że ten element bufora staje się pusty. Po wykonaniu odczytu z ostatniego elementu bufora, następny odczyt będzie wykonywany z pierwszego elementu bufora.

Własność bezpieczeństwa

konsument nie wykona odczytu z pustego elementu bufora i producent nie wykona zapisu do niepustego elementu bufora

Własność żywotności

konsument zawsze wykona odczyt danych z bufora i producent zawsze wykona zapis danych do bufora.

Wniosek

Jeżeli w problemie producent-konsument własność bezpieczeństwa jest spełniona to wszystkie dane wyprodukowane przez producentów zostaną skonsumowane przez konsumentów.

przykład 1:

x ←0

p1: x ←1	q1: wypisz x
p2: x ←2	q2: wypisz x
p3: koniec	q3: koniec

Przedstawiony przykład pokazuje problem producenta (proces *p*) i konsumenta (proces *q*) z buforem jednoelementowym (zmienna *x*).

scenariusze:

a) q1-q2-p1-p2-p3-q3 (wynik: 00)

b) p1-q1-q2-p2-p3-q3 (wynik: 11)

c) p1-p2-q1-q2-p3-q3 (wynik: 22)

d) p1-q1-p2-q2-p3-q3 (wynik: 12)

ćwiczenie 1:

zsynchronizować procesy *p* i *q* w taki sposób, aby zawsze realizował się scenariusz (d) .

rozwiązanie:

x ←0

semafor sp ← 1, sq ←0

wait (sp)	***wait(sq)***
p1: x ←1	q1: wypisz x
signal (sq)	***signal(sp)***
wait (sp)	***wait(sq)***
p2: x ←2	q2: wypisz x
signal (sq)	***signal(sp)***
p3: koniec	q3: koniec

uwaga: operacje na semaforach nie są etykietowane dla przejrzystości programu

W tym programie możliwy jest tylko scenariusz: p1-q1-p2-q2

Ogólne rozwiązanie problemu producent konsument przy pomocy semaforów

1. Przypadek z nieskończonym buforem

bufor [1.. ∞ ⎤

i ← 1, j ← 1

Semafor pełny ← 0

Semafor można_pisać ← 1, można_czytać ← 1

p0:	powtarzaj	q0:	powtarzaj
p1:	dane ← produkuj	q1:	wait **(pełny)**
p2:	wait (można_pisać)	q2:	wait (można_czytać)
p3:	bufor [i] ← dane	q3:	dane ← bufor[j]
p4:	i ← i + 1	q4:	j ← j + 1
p5:	signal (można_pisać)	q5:	signal (można_czytać)
p6:	signal **(pełny)**	q6:	konsumuj(dane)

Zakłada się, że istnieje skończona liczba procesów *p* i procesów *q*. Zmienna *i* jest współdzielona przez procesy producenta (p), a zmienna *j* jest współdzielona przez procesy konsumenta (q)

Zauważ że: zapis i odczyt danych na buforze realizują odpowiednio instrukcje p3..p4 lub q3..q4, które są sekcjami krytycznymi, w przypadku rywalizacji wielu procesów zapisujących lub czytających dane. Stąd, dla zapewnienia wzajemnego wykluczania zapisu bądź odczytu z bufora użyto semaforów binarnych *można_pisać* i *można_czytać* (dlaczego dwa różne semafory?).

Pytanie: kiedy można pominąć semafory binarne *można_pisać* i *można_czytać* ?

2. Przypadek ze skończonym buforem

bufor [1.. N]

i ← 1, j ← 1

Semafor pełny ← 0, pusty ← N

Semafor można_pisać ← 1, można_czytać ← 1

p0:	powtarzaj	q0:	powtarzaj
p1:	dane ← produkuj	**q1:**	wait **(pełny)**
p2:	wait **(pusty)**	q2:	wait (można_czytać)
p3:	wait (można_pisać)	q3:	dane ← bufor[j]
p4:	bufor [i] ← dane	q4:	j ← j mod N + 1
p5:	i ← i mod N + 1	q5:	signal (można_czytać)
p6:	signal (można_pisać)	**q6:**	signal **(pusty)**
p7:	signal **(pełny)**	q7:	konsumuj (dane)

Zauważ :

a) instrukcje *p5, q4* pozwalają poruszać się cyklicznie po elementach bufora

b) instrukcje *p4,p5* lub *q3,q4* są sekcjami krytycznymi dla swojej grupy procesów,

c) semafor *pełny* zapewnia, że dane nie będą czytane przed ich zapisaniem w buforze

d) semafor *pusty* zapewnia, że dane zapisane nie będą nadpisane

e) jakimi wartościami są inicjalizowane semafory *pełny* i *pusty*

Problem czytelników i pisarzy

Definicje

czytelnia – szeroko rozumiana baza danych

czytelnicy – procesy wykonujący odczyt danych w czytelni

pisarze – procesy wykonujący zapis danych w czytelni

Założenie

Czytelnicy i pisarze nieskończenie wiele razy mogą wykonywać sekcję lokalną i wchodzić do czytelni. Wejście do czytelni oznacza możliwość wykonania zapisu bądź odczytu danych. Procesy mogą utknąć w swoich sekcjach lokalnych tzn. zrezygnować z wchodzenia do czytelni. Procesy będące w czytelni muszą ją w końcu opuścić (czytelnia wykonuje się z postępem).

Własność bezpieczeństwa

Jeżeli w czytelni jest pisarz to pozostałe procesy muszą być poza czytelnią

Własność żywotności

Każdy proces po opuszczeniu sekcji lokalnej wejdzie do czytelni

Uwaga: własność bezpieczeństwa dopuszcza do jednoczesnego korzystania z czytelni przez wielu czytelników

Problem synchronizacji czytelników i pisarzy rozwiązuje się przy pomocy narzędzia zwanego monitorem.

MONITOR

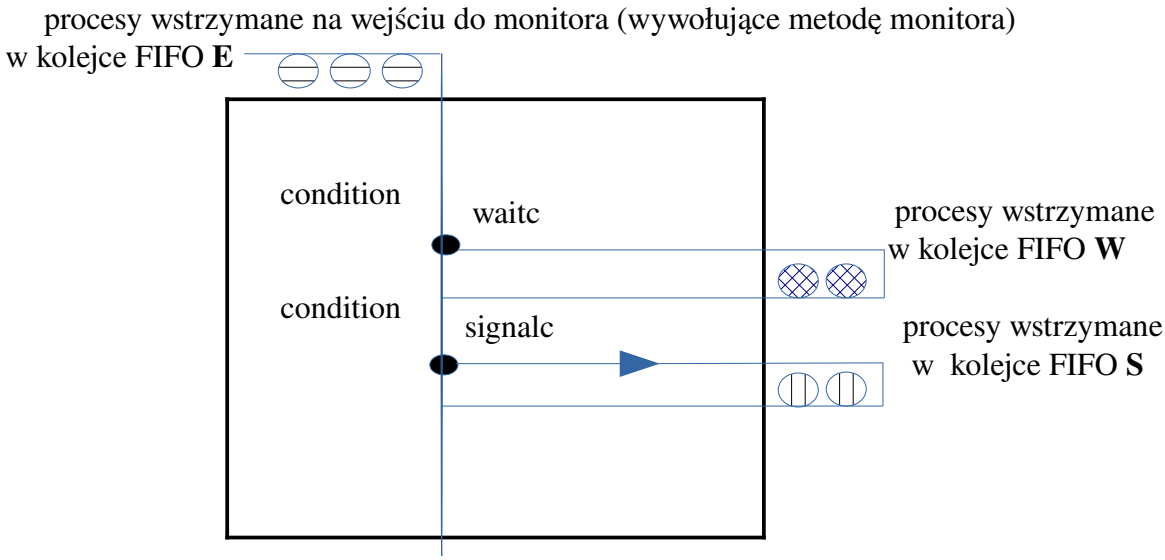
Monitor jest mechanizmem synchronizacyjnym będący obiektem, w którym deklaracje zmiennych i operacje są kapsułkowane w postaci klasy znanej z programowania obiektowego. Wymaga się, aby wszystkie operacje na monitorze wykonywały się ze **wzajemnym wykluczaniem.** Dodatkowym wymaganiem jest, żeby wszystkie pola (składowe) monitora były prywatne. Te wymagania gwarantują, że monitor będący obiektem współdzielonym przez procesy zachowuje spójność danych swoich pól.

Do synchronizacji procesów wykorzystuje się zmienne warunkowe, które są częścią składową monitorów. Zmienne warunkowe pozwalają wstrzymywać lub wznawiać procesy w zależności od warunków logicznych charakteryzujących stan monitora.

Określenia:

- proces wchodzi do monitora oznacza, że rozpoczął wykonywanie operacji (metody) na monitorze
- proces jest w monitorze oznacza, że jest w trakcie wykonywania operacji (metody) na monitorze

schemat wykonania operacji na monitora:



Uwaga: W monitorze klasycznym zwanym również monitorem z wymaganiem natychmiastowego wznowienia (WNW) priorytety kolejek spełniają następujący warunek: **E < S < W.** Warunek WNW oznacza, że sygnalizowany proces z kolejki **W** zawsze pierwszy opuści monitor. Proces, który wykonał operację *signalc* zostaje wstrzymany w kolejce **S** dopóki proces z kolejki **W** nie opuści monitora. W następnej kolejności proces z koleki **S**, opuszcza monitor, a na końcu ewentualnie proces z kolejki **E** na wejściu do monitora. Wyjątkiem jest sytuacja, gdy operacja *signalc* jest ostatnią instrukcją w metodzie monitora. W takim przypadku proces nie zostaje zatrzymany w kolejce *S* tylko od razu opuszcza monitor.

Zmienne warunkowe

Definicja

Zmienne warunkowe są to obiekty reprezentujące warunki logiczne dla których zdefiniowano atomową operację *waitc* służącą do wstrzymania procesu jeżeli oczekiwany warunek logiczny jest nieprawdziwy oraz atomową operację *signalc* do wznowienia procesu jeżeli warunek logiczny jest prawdziwy. Powiązanie zmiennej warunkowej z odpowiednim warunkiem logicznym dokonuje się najczęściej przy pomocy zewnętrznej instrukcji warunkowej w rodzaju: *if (warunek_logiczny) waitc(zmienna_warunkowa)*. Wstrzymywane procesy są umieszczane w kolejkach FIFO związanych z obiektem.

Operacje na zmiennej warunku

deklaracja zmiennej warunku: **condition** W

operacja wstrzymania procesu:

waitc (W) :

W.FIFO ← proces

proces.stan ← wstrzymany

monitor.blokada ← zwolnij

operacja wznowienia wstrzymanego procesu:

signalc (W) :

if (W.FIFO ≠ ∅)

proces ← W.FIFO

proces.stan ← gotowy

operacja sprawdzenia stanu zmiennej warunku

empty (W) :

if (W.FIFO = ∅)

return true

else

return false

Obiekty chronione

Obiekty chronione podobnie jak obiekty klasycznego monitora spełniają wymagania wzajemnego wykluczania operacji na obiekcie oraz prywatności jego pól składowych. Jednak nie korzysta się w nich ze zmiennych warunkowych. Synchronizacja procesów odbywa się tutaj na początku (wejściu) i końcu (wyjściu) każdej operacji. Wejście do operacji może być opatrzone klauzulą zwaną dozorem, który jest wyrażeniem logicznym zapisanym przy pomocy składowych pól obiektu. W przypadku gdy wartość tego wyrażenia jest fałszywa proces zostaje wstrzymany w kolejce FIFO. Z każdym wejściem (dozorem) jest związana osobna kolejka FIFO. Z kolei przy wyjściu z danej operacji obliczane są ponownie wszystkie dozory i w przypadku wartości true odpowiedni proces zostaje wznowiony i wchodzi do danej operacji. Jeżeli w kolejce związanej z określonym dozorem czeka więcej procesów i jeżeli wartość tego dozoru będzie pozostawać niezmiennie true to wszystkie te procesy po kolei zostaną wznowione bez przełączenia kontekstu do innej kolejki związanej z innym dozorem równym true. To zwiększa efektywność operacji na obiektach chronionych. Dla obiektów chronionych zasadę pierwszeństwa wznawianych procesów przedstawia warunek **E < W**, gdzie **E** jest kolejką FIFO procesów rywalizujących o dostęp do operacji na obiekcie, a **W** kolejką FIFO związaną z dozorowanym wejściem operacji.

Obiekty chronione bardzo ułatwiają programowanie monitorów. Tego typu obiekty są zaimplementowane w języku ADA.

Przykłady zastosowania monitora

1. Problem wzajemnego wykluczania

definicja monitora:

```
Monitor  SK
        operation wykonaj
                sekcja_krytyczna()     // funkcja wykonująca sekcję krytyczną
```

Program:

SK s

p0:Powtarzaj	q0:Powtarzaj
p1: sekcja_lokalna	q1: sekcja_lokalna
p2: s.wykonaj	q2: s.wykonaj

Sekcja krytyczna jest wykonywana w instrukcjach *p2* i *q2,* a wzajemne wykluczanie gwarantuje operacja wykonaj ponieważ jest metodą monitora.

2. Problem producenta i konsumenta

definicja monitora:

```
Monitor PK
        bufor_typ bufor
        condition  niepełny, niepusty

        operation  wstaw (dane)
                        if bufor.pełny = true
                                waitc (niepełny )
                        bufor.zapis(dane)
                        signalc( niepusty)

        operation  pobierz (dane)
                        if bufor.pusty = true
                                waitc (niepusty )
                        bufor.odczyt(dane)
                        signalc( niepełny)
```

Program:

```
                        PK Bufor
p1: powtarzaj                           q1: powtarzaj
p2:     dane ← produkuj         q2:     Bufor.pobierz (dane)
p3:     Bufor.wstaw( dane )             q3:     wykonaj (dane )
```

Uwaga: Powyższe definicje zakładają że mamy zdefiniowaną klasę bufor_typ, która ma zdefiniowane metody: *odczyt, zapis* realizujące odczyt i zapis cykliczny na tablicy oraz metody: pusty i pełny które sprawdzają ilość danych znajdujących się w tablicy.

3. Problem czytelników i pisarzy

definicja monitora:

```
Monitor  CP

    condition  można_czytać, można_pisać

    int czyta← 0 ,  pisze← 0 , chce_czytać← 0 , chce_pisać ← 0


    operation początek_czytania
    [1]     if ( pisze =1 lub  chce_pisać >0 )
    [2]                         chce_czytać← chce_czytać +1
    [3]                         waitc (można_czytać)
    [4]                         chce_czytać← chce_czytać -1
    [5]     czyta ← czyta +1
    [6]     signalc(można_czytać)

    operation początek_pisania
    [1]     if  ( czyta > 0 lub pisze = 1)
    [2]                 chce_pisać ←  chce_pisać +1
    [3]                 waitc (można_pisać)
    [4]                 chce_pisać ←  chce_pisać -1
    [5]     pisze ← 1

    operation koniec_czytania
    [1]     czyta ← czyta -1
    [2]     if  czyta = 0
    [3]                 signalc(można_pisać)

    operation koniec_pisania
    [1]     pisze ← 0
    [2]     if ( chce_czytać > 0 )
    [3]           signalc (można_czytać)
    [4]     else
    [5]           signalc (można_pisać)
```

Własność bezpieczeństwa: $(\text{czyta} > 0 \Rightarrow \text{pisze}=0) \wedge (\text{pisze} \leq 1) \wedge (\text{pisze}=1 \Rightarrow \text{czyta}=0)$

Uwaga:

Warunek w operacji *początek_czytania* zapobiega zagłodzeniu pisarzy w przypadku gdyby czytelnicy jeden po drugim nieustannie chcieli wchodzić do czytelni. Warunek w operacji *koniec_pisania* zapobiega zagłodzeniu czytelnika. Zauważ że kolejni czytelnicy będą zwalniani przez poprzedniego zwolnionego czytelnika w operacji *początek_czytania*.

Definicja obiektu chronionego do synchronizacji czytelników i pisarzy:

```
Protected CP
        int czyta ← 0,  pisze  ← 0
        operation początek_czytania when pisze=0
                czyta ← czyta+1
        operation koniec_czytania
                czyta  ←  czyta-1
        operation początek_pisania when pisze=0 i czyta=0
                        pisze  ←  1
        operation koniec_pisania
                        pisze ←  0
```

Ten obiekt nie zapewnia własności żywotności (dlaczego?).

Program:

CP czytelnia

p1: powtarzaj		q1: powtarzaj	
p2:	czytelnia. początek_pisania()	q2:	czytelnia. początek_czytania()
p3:	zapis (dane)	q3:	odczyt(dane)
p4:	czytelnia. koniec_pisania()	q4:	czytelnia. koniec_czytania()

Procesy *p* reprezentują pisarzy a procesy *q* czytelników.

Ćwiczenie 1: Napisać w pseudokodzie implementację obiektu chronionego do rozwiązania problemu czytelników i pisarzy bez możliwości zagłodzenia procesów.

Protected CP

```
        int czyta ← 0,  pisze  ← 0
        operation początek_czytania
        operation wpuść_czytelnika
        operation koniec_czytania
        operation początek_pisania
        operation wpuść_pisania
        operation koniec_pisania
```

Ćwiczenie2: Napisz w pseudokodzie definicję monitora, który będzie symulował działanie semafora

Ćwiczenie3: Napisz w pseudokodzie definicję monitora, który będzie symulował działanie bariery. Działanie bariery polega na tym, że proces '*dochodząc*' do bariery wykonuje operację monitora *waitb* i zostaje na niej zatrzymany, przy czym monitor przechowuje liczbę wstrzymanych procesów. Gdy ona osiągnie wartość progową (którą ustala się dla każdej bariery podczas inicjalizacji), to wszystkie wstrzymane procesy zostają wznowione. W praktyce ostatni proces, który osiągnie próg bariery nie jest wstrzymywany tylko inicjuje zwolnienie wstrzymanych procesów. Po zwolnieniu wszystkich procesów bariera zostaje zresetowana tzn. działa tak jak na początku po inicjalizacji obiektu.

Uwagi końcowe:

Implementacje monitorów wyglądają różnie w różnych językach programowania. W języku C++ z biblioteką pthread monitor implementuje użytkownik wykorzystując zmienne warunku typu *pthread_cond_t* oraz *pthread_mutex_t*. W języku Jawa używa się klauzuli *synchronized* przy definicji metod obiektów oraz operacji *wait* i *notify*. Zarówno monitory implementowane przy pomocy biblioteki *pthread* jak i te w języku Java na ogół nie spełniają warunku **WNW** chyba, że użytkownik zada sobie więcej trudu i sam zaimplementuje odpowiednio monitor. Do ochrony metod monitorów używa się mutexów, na których wstrzymywane wątki nie oczekają w kolejkach tylko są arbitralnie wznawiane. Dlatego najczęściej implementuje się monitory, dla których warunek priorytetów kolejek wznawianych procesów można zapisać następująco: $E = W \prec S$. Kolejka ***S*** dla procesów wykonujących *signalc,* w ogóle nie istnieje i taki proces jako pierwszy musi opuścić metodę monitora. Z kolei procesy wstrzymane na zmiennej warunku ***W*** po wznowieniu rywalizują o wejście do monitora z procesami wstrzymanymi na mutexach ***E*** chroniących monitor. Tego typu monitory są bardziej złożone, gdyż po opuszczeniu monitora przez proces nie można przewidzieć, którym wejściem (E czy W) do monitora wejdzie następny proces. Przykładowo, powyżej zdefiniowany w pseudokodzie monitor czytelników i pisarzy nie będzie działał poprawnie, gdy nie spełnimy warunku natychmiastowego wznowienia. (dlaczego?)