Podstawy programowania współbieżnego
II

PROBLEMY SYNCHRONIZACYJNE I NARZĘDZIA SYNCHRONIZACJI

Omawiane zagadnienia dotyczą zapewnienia poprawności wykonywania się programów współbieżnych w sytuacjach, gdy procesy współdziałają bądź rywalizują ze sobą. Zachowanie własności bezpieczeństwa i żywotności (patrz wykład 1) jest zapewnione przez odpowiednią synchronizację procesów, która powinna wyeliminować niekorzystne scenariusze wykonania programu współbieżnego. Synchronizacji procesów dokonuje się przy pomocy odpowiednich algorytmów bądź wysokopoziomowych narzędzi synchronizacyjnych wspieranych przez różne biblioteki programistyczne używane w programowaniu współbieżnym. Można wyróżnić trzy podstawowe typy problemów wymagających synchronizacji procesów, tzw. klasyczne problemy synchronizacyjne. Celem wykładu jest omówienie klasycznych problemów synchronizacyjnych oraz narzędzi synchronizacji: mutex,semafor, monitor.

Problem wzajemnego wykluczania

Problem wzajemnego wykluczania pojawia się w sytuacji gdy procesy rywalizują o dostęp do wspólnego zasobu, który powinien być użytkowany w danym momencie wyłącznie przez jeden proces. Taka sytuacja może wystąpić bardzo często gdy procesy próbują jednocześnie zapisywać wspólny obszar pamięci. Przykładem mogą tu być instrukcje, w których zmienne mają podwójne wystąpienia krytyczne (np. instrukcja *n*++ w języku C).

Ogólne sformułowanie problemu

Szereg procesów wykonuje następujący fragment kodu:

```
p1: powtarzaj
p2:     sekcja_lokalna
p3:     sekcja_krytyczna
```

Założenia

a) procesy wykonują nieskończenie wiele razy sekcję lokalną i krytyczną

b) sekcja krytyczna wykonuje się z postępem (proces nie może się tam zakończyć)

c) sekcja lokalna nie musi wykonywać się z postępem (proces może jej nie opuścić)

Własność bezpieczeństwa – Sekcja krytyczna może być wykonywana tylko przez jeden proces.

Własność żywotności – każdy proces, który opuści sekcję lokalną wykona sekcję krytyczną

Ogólne rozwiązanie problemu

```
p1: powtarzaj
p2:     sekcja_lokalna
p3:     protokół_wstępny
p4:     sekcja_krytyczna
p5:     protokół_końcowy
```

Uwagi:

a) sekcja lokalna jest wykonywana współbieżnie (w przeplocie)

b) sekcja krytyczna wykonuje się sekwencyjnie (bez przeplotu)

c) protokół wstępny i protokół końcowy to dodatkowe fragmenty kodu, które mają zapewnić właściwą synchronizację procesów.

Algorytmy i narzędzia synchronizacji procesów w problemie wzajemnego wykluczania

a) algorytm Dekkera

chcep ← chceq ← false

czyja_kolej ←1

powtarzaj		powtarzaj	
p1:	sekcja_lokalna	q1:	sekcja_lokalna
p2:	chcep ← true	q2:	chceq ← true
p3:	while (chceq=true)	q3	while (chcep=true)
p4	if czyja_kolej=2	q4:	if czyja_kolej=1
p5	chcep ← false	q5:	chceq ← false
p6:	czeka_aż (czyja_kolej=1)	q6:	czeka_aż (czyja_kolej=2)
p7:	chcep ← true	q7:	chceq ← true
p8:	sekcja_krytyczna	q8:	sekcja_krytyczna
p9:	czyja_kolej ← 2	q9:	czyja_kolej ← 1
p10:	chcep ← false	q10:	chceq ← false

Algorytm zapewnia spełnienie własności bezpieczeństwa i żywotności. Zauważ że proces może utknąć w sekcji lokalnej a inny proces może wielokrotnie odwiedzać sekcję krytyczną. Wszystkie instrukcje zawierają co najwyżej pojedyncze wystąpienia krytycznie. Algorytm jest słuszny tylko dla dwóch procesów.

Dowód poprawności:

Definicje:

Założenie: $s_1,s_2,\ldots,s_i,\ldots$ - ciąg stanów programu współbieżnego zwany obliczeniem

X – formuła *X* jest prawdziwa w stanie s_i

~X – formuła *X* jest nieprawdziwa w stanie s_i

1) $\Box A$ - formuła jest prawdziwa w stanie s_i danego obliczenia wtedy i tylko wtedy, gdy formuła A jest prawdziwa w wszystkich stanach s_j dla $j \geq i$ tego obliczenia

2) $\Diamond A$ - formuła jest prawdziwa w stanie s_i danego obliczenia wtedy i tylko wtedy, gdy formuła A jest prawdziwa w stanie s_j dla $j \geq i$

3) $\Diamond\Box A$ -formuła jest prawdziwa w stanie s_i danego obliczenia wtedy i tylko wtedy, gdy formuła A jest prawdziwa w wszystkich stanach s_j dla $j \geq k$ tego obliczenia, gdzie $k \geq i$

wł. bezpieczeństwa: $\Box\sim(p8 \wedge q8)$

wł. żywotności: $\Box (p2 \Rightarrow \Diamond p8) \wedge \Box (q2 \Rightarrow \Diamond q8)$

niezmienniki

1) czyja_kolej=1 $\vee$ czyja_kolej=2

2) p3..p5 $\vee$ p8..p10 $\Leftrightarrow$ chcep

3) q3..q5 $\vee$ q8..q10 $\Leftrightarrow$ chceq

Dowód spełnienia własności bezpieczeństwa:

własność bezpieczeństwa oznacza tutaj, że ~(p8^q8) jest niezmiennikiem
dowodzimy, że nie istnieje stan dla którego zachodzi p8^q8

1) formuła p8^q8 jest fałszywa w pierwszym stanie i we wszystkich stanach dla których ~p8 $\vee$ ~q8

2) p8^q8 może zachodzić wtedy, gdy istnieją stany, dla których (p8$\wedge$ q3 $\wedge$ $\sim$chcep) jest prawdziwe lub (q8 $\wedge$ p3 $\wedge$ $\sim$chceq) jest prawdziwe.

Na podstawie niezmiennika (2) mamy (p8$\wedge$ q3 $\wedge$ $\sim$chcep) $\Leftrightarrow$ (chcep $\wedge$ q3 $\wedge$ $\sim$chcep)

formuła (chcep $\wedge$ q3 $\wedge$ $\sim$chcep) jest zawsze fałszywa, czyli nie może istnieć taki stan, dla którego zachodzi p8$\wedge$ q3 $\wedge$ $\sim$chcep.

Dowód własności żywotności:
sprawdzamy czy proces p może być zagłodzony tzn. zakładamy że istnieje stan dla którego zachodzi: p2 $\Rightarrow$ ~$\Diamond$p8

zał. $\Box$ czyja_kolej=2 to p2 $\Rightarrow$ $\Diamond\Box$p6 $\Leftrightarrow$ $\Diamond\Box$~chcep $\Rightarrow$ $\Diamond$ q9 $\Rightarrow$ $\Diamond$ $\Box$ czyja_kolej =1
otrzymaliśmy sprzeczność z założeniem, stąd wynika że proces *p* nie utknie na zawsze w instrukcji p6. Więc formuła p2 $\Rightarrow$ $\Diamond\Box$p6 jest fałszywa. (wniosek 1)

zał. $\Box$ czyja_kolej = 1 bo proces *q* nie zmieni wartości *czyja_kolej*

zatem zachodzi formuła p2 $\Rightarrow$ $\Diamond\Box$ p3..p4
ale wtedy proces *q* musi utknąć na zawsze w sekcji lokalnej lub w instrukcji *q6.*

Zatem zachodzi: ($\Diamond\Box$q1 $\vee$ $\Diamond\Box$q6), a wtedy na podstawie niezmiennika (3) zachodzi $\Diamond\Box$~chceq.
Stąd wynika, że proces p nie może utknąć na zawsze w instrukcjach p3..p4. Więc formuła p2 $\Rightarrow$ $\Diamond\Box$ p3..p4 jest fałszywa. (wniosek 2)

Z wniosków (1) i (2) wynika, że p2 $\Rightarrow$ ~$\Diamond$p8 nie zachodzi, a więc proces *p* nie może być zagłodzony.

Ćwiczenie1:

czyja_kolej ←1

Powtarzaj	Powtarzaj
p1: sekcja_lokalna	q1: sekcja_lokalna
p2: while (czyja_kolej=2)	q2: while (czyja_kolej=1)
p3: sekcja_krytyczna	q3: sekcja_krytyczna
p4: czyja_kolej ← 2	q4: czyja_kolej ←1

Czy powyższy program zapewnia spełnienie własności bezpieczeństwa i żywotności?

Ćwiczenie2:

chcep ← chceq ← false

Powtarzaj	Powtarzaj
p1: sekcja_lokalna	q1: sekcja_lokalna
p2: chcep ←true	q2: chceq ←true
p3: while (chceq=true)	q3: while (chcep=true)
p4: sekcja_krytyczna	q4: sekcja_krytyczna
p5: chcep ←false	q5: chceq ←false

Ćwiczenie3:

chcep ← chceq ← false

Powtarzaj	Powtarzaj
p1: sekcja_lokalna	q1: sekcja_lokalna
p2: chcep ←true	q2: chceq ←true
p3: while (chceq=true)	q3: while (chcep=true)
p4: chcep ←false	q4: chceq ←false
p5: chcep ←true	q5: chceq ←true
p6: sekcja_krytyczna	q6: sekcja_krytyczna
p7: chcep ←false	q7: chceq ←false

Czy powyższe programy zapewnia spełnienie własności bezpieczeństwa i żywotności ? Jeżeli nie to udowodnić pokazując odpowiedni przeplot.

b) algorytm piekarniany

```
                    numer_biletu[1..n] ← 0
powtarzaj
p1:sekcja_lokalna
p2: numer_biletu[i] ← max(numer_biletu) +1
p3: for j ←1,n
p4:     czekaj_na ( numer_biletu[j]=0 lub numer_biletu[i] << numer_biletu[j])
p5: sekcja_krytyczna
p6: numer_biletu[i] ← 0
```

Algorytm jest wykonywany przez skończoną liczbę procesów. Procesy są numerowane wartościami *1,2,3...n*. Powyższy algorytm jest wykonywany przez proces o numerze *i*.

Wyrażenie *numer_biletu[i]* ≺≺ *numer_biletu[j]* jest prawdziwe jeżeli:
a) *numer_biletu[i]* ≺ *numer_biletu[j])*
b) *i* ≺ *j* gdy *numer_biletu[i]* = *numer_biletu[j]*.
Zauważ że może się zdarzyć że procesy będą mieć jednakowe numery biletów.

Algorytm rozstrzyga pierwszeństwo wejścia do sekcji krytycznej w przypadku gdy rywalizuje o nie wiele procesów. Algorytm zapewnia spełnienie własności bezpieczeństwa (jednoznaczny warunek w czekaj_na) i własności żywotności (instrukcja *p2*).

Ćwiczenie 4:
Pokazać na przeplocie że numery biletów mogą rosnąć w nieskończoność

c) semafor

Semafor jest wysokopoziomowym narzędziem synchronizacji procesów, który pozwala kontrolować liczbę procesów wykonujących jednocześnie dany fragment programu współbieżnego.

Definicja klasyczna semafora

Semafor jest obiektem przechowującym wartości całkowite nieujemne, na którym są wykonywane operacje atomowe *wait* i *signal*.

Składowe semafora S:

S.V – wartość całkowita

S.L – zbiór procesów wstrzymanych

Operacje na semaforze:

semafor S ← *wartość_całkowita* (deklaracja i inicjalizacja semafora)

```
wait ( S )                          (definicja operacji)
    if ( S.V = 0 )
        proces.stan ← wstrzymany
        S.L ∪ { proces }
    else
        S.V ← S.V – 1
```

Jeżeli proces wykonuje operację *wait* na semaforze o wartości zero to taki proces nie zakończy tej operacji tylko przejdzie w stan *wstrzymany* i zostanie zapamiętany w zbiorze semafora. Jeżeli wartość semafora będzie większa od zera to proces wykona operację *wait* skutkiem czego wartość semafora zostanie obniżona o jeden. Procesy wstrzymane są zapamiętane w zbiorze. Jeżeli zbiór semafora nie jest pusty to wartość semafora musi być równa zero.

Przykład:

semafor s ← 1

p1: wait (s)	q1: wait (s)
p2: exit	q2: exit

Przy scenariuszu: p1-p2 proces p zakończy działanie, a proces *q* pozostanie wstrzymany na instrukcji q1. Czy istnieje scenariusz, przy którym oba procesy kończą działanie?

```
signal ( S )                    (definicja operacji)
        if ( S.L = ⊘ )
        S.V ← S.V +1
        else
        proces.stan ← gotowy
        S.L - { proces }
```

Jeżeli proces wykona operację *signal* na semaforze, na którym zostały wstrzymane inne procesy, to wtedy jeden z nich (proces losowo wybrany) zostanie wznowiony tzn. przejdzie w stan *gotowy*. Wartość semafora pozostanie wówczas równa zero. Gdy semafor nie posiada procesów wstrzymanych to skutkiem wykonania operacji *signal* jest zwiększenie wartości semafora o jeden. Wykonanie sekwencji operacji *wait* i *signal* nie zmienia wartości semafora.

oznaczenia:
k – wartość początkowa semafora
#wait(S) – liczba wykonanych operacji *wait* na semaforze
#signal(S) – liczba wykonanych operacji *signal* na semaforze
S.V – wartość semafora

Niezmienniki semafora:

1) $S.V \geq 0$
2) $S.V = k - \#wait(S) + \#signal(S)$

Przykład:

semafor s ← 0

p1: wait (s)	q1: signal (s)
p2: exit	q2: exit

Przy scenariuszu: q1-p1-q2-p2 wartość semafora będzie równa zero. Wynika to bezpośrednio z niezmiennika (2). Scenariusz p1-q1-p2-q2 jest niepoprawny ponieważ instrukcja p1 jako pierwsza nie może być wykonana (proces p zostanie wstrzymany).

Ćwiczenie 5

semafor s1 ← 0, s2 ← 0

p1: wait (s1)	q1: wait (s2)	r1: signal(s1)
p2: wypisz p	q2: wypisz q	r2: signal(s2)
p3:exit	q3:exit	r3:exit

Podać scenariusze dające różne wyniki (wypisane) tego programu współbieżnego

Synchronizacja procesów w problemie wzajemnego wykluczania przy pomocy semafora

semafor s ← 1

powtarzaj	powtarzaj	powtarzaj
p1:sekcja_lokalna	q1:sekcja_lokalna	r1:sekcja lokalna
p2: wait (s)	q2: wait (s)	r2:wait(s)
p3: sekcja_krytyczna	q3: sekcja_krytyczna	r3:sekcja krytyczna
p4: signal(s)	q4: signal(s)	r4:signal(s)

Z konstrukcji programu wnioskujemy że liczba procesów w sekcji krytycznej (#sk) wynosi:
#sk=#wait-#signal
Z niezmiennika semaforowego (2) otrzymujemy:
S.V=1-#wait+#signal = 1 - #sk
Na podstawie niezmiennika semaforowego (1) mamy:

$S.V=1-\#sk \geq 0$

stąd liczba procesów w sekcji krytycznej: $\#sk \leq 1$

Wniosek: Program spełnia własność bezpieczeństwa

Zauważ że:

Liczba procesów w sekcji krytycznej zależy od wartości początkowej semafora. Wzajemne wykluczanie gwarantuje semafor z wartością początkową równą 1. W powyższym kodzie semafor przyjmuje wartości 0 lub 1. Taki semafor nazywamy semaforem binarnym.

Semafor binarny gwarantuje wzajemne wykluczanie w sekcji krytycznej w przypadku rywalizacji dowolnej liczby procesów. Zagłodzenie jakiegoś procesu teoretycznie jest możliwe ponieważ operacja *signal* zwalnia losowo wybrany proces. Niedopuszczenie do zagłodzenia gwarantuje semafor silny. W takim semaforze zatrzymane procesy są pamiętane w kolejce FIFO i co za tym idzie mogą być wznawiane w odpowiedniej kolejności.

Przykłady z wykorzystaniem semaforów

Przykład 1: kontrola liczby procesów w programie współbieżnym

```
                        semafor s ← maxw
p1: Powtarzaj
p2:     wait(s)                     w1: wykonaj zadanie
p3:     utworz_proces(w)            w2: signal(s)
```

W przykładzie proces *p* tworzy procesy, których liczba jest kontrolowana przez semafor *s*. Maksymalna liczba zadań wykonywanych współbieżnie wynosi *maxw*.

Przykład 2: Sprawdzanie stanu zakończenia obliczeń w programie współbieżnym

```
                        semafor s ← 0
                        x ← y ← 0

p1: wait(s)             q1: x ← zadanie 1       r1: y ← zadanie 2
p2: wait(s)             q2: signal(s)           r2: signal(s)
p3: oblicza x+y
```

W przykładzie proces *p* oblicza sumę *x*+*y* gdzie wartości *x* i *y* są obliczane współbieżnie przez inne procesy. Aby wynik końcowy był poprawny proces *p* musi znać końcowe wartości *x* i *y*.

Ćwiczenie 6:

```
                        semafor b ← 0, s ← 1
                        m ← ?

p1: wait (s)
p2: m ← m -1
p3: if m= 0
p4:     sgnal (b)
    else
p5:     signal (s)
p6: wait (b)
p7: signal(b)
p8: koniec
```

Program jest wykonywany współbieżnie przez cztery procesy p,q,r,s odpowiednio. Podać scenariusze dla tego programu w zależności od wartości początkowej *m > 0*

d) mutex

Mutex jest narzędziem synchronizacji dedykowanym do zabezpieczania sekcji krytycznych w problemie wzajemnego wykluczania. Ma zdefiniowane dwie operacje atomowe: *lock* i *unlock*. Jeżeli proces chce wykonać sekcję krytyczną musi wpierw zdobyć klucz (operacja *lock*). Po wykonaniu sekcji krytycznej proces zwraca klucz (operacja *unlock*). Klucz może być w posiadaniu tylko jednego procesu. Próba zdobycia klucza, który jest w posiadaniu innego procesu kończy się wstrzymaniem danego procesu (przechodzi w stan *wstrzymany*). Istnienie jednego klucza gwarantuje wzajemne wykluczanie w sekcji krytycznej.

Mutex jest rodzajem semafora binarnego z tą różnicą, że operacja unlock powinna być zawsze wykonywana po operacji lock.

Przykład zastosowania mutexu dla ochrony sekcji krytycznej w programie współbieżnym

mutex m

powtarzaj	powtarzaj
p1:sekcja_lokalna	q1:sekcja_lokalna
p2: lock (m)	q2: lock (m)
p3: sekcja_krytyczna	q3: sekcja_krytyczna
p4: unlockl(m)	q4: unlockl(m)

Zadanie: problem 5 filozofów

1) Problem synchronizacyjny znany jako problem 5 filozofów. Każdy z pięciu filozofów wykonuje dwie czynności: je i myśli. Czynności te są wykonywane w nieskończoność. Filozofowie siedzą przy okrągłym stole. Na stole jest 5 widelców. Każdy filozof może jeść jeżeli posiada dwa widelce. Widelec nie może być w posiadaniu jednocześnie przez więcej niż jednego filozofa. Napisać program który symuluje działanie filozofów. Działanie każdego filozofa wykonuje funkcja odrębnego wątku. Odpowiedni wydruk programu powinien pokazać poprawną synchronizację.

Własność bezpieczeństwa:
- nie dopuścić do zakleszczenia
- nie dopuścić do sytuacji by dwóch filozofów posiadało jednocześnie ten sam widelec

Własność żywotności:
- nie dopuścić do sytuacji w której filozof będzie zagłodzony

przykładowy proces filozofa:

```
p1: powtarzaj
p2:     myślenie
p3:     wez_widelec(prawy)
p4:     wez_widelec(lewy)
p5:     jedzenie
p6:     oddaj_widelec(prawy)
p7:     oddaj_widelec(lewy)
```