1 Klasyczne problemy współbieżności w asynchronicznym modelu komunikacyjnym

Problem 1 Wzajemne wykluczanie. Stosujemy notację z wykładu oraz zakładamy, że bufor jest nieograniczony.

Pomysł polega na umieszczeniu początkowo w buforze jednego egzemplarza komunikatu `Bilet`. Proces wejdzie do sekcji krytycznej o ile w buforze jest komunikat, w przeciwnym razie będzie czekał. Wchodząc do sekcji krytycznej oczywiście wyjmuje komunikat z buforu a umieszcza go w nim zwalniając sekcję.

```
type
  Komunikaty = (Bilet)
var
  b: buffer;

process P (i: integer);
var
  k: Komunikaty;
begin
  repeat
    sekcja_lokalna;
    GetMessage (b, k)
    rejon_krytyczny;
    SendMessage (b, k)
  until false
end;

begin
  SendMessage (b, Bilet);
  cobegin P(1); P(2) coend
end
```

Zwróćmy uwagę, że dla poprawności tego rozwiązania kluczowa jest żywotność operacji pobrania z bufora. Ponieważ pracujemy w modelu rozproszonym zmienne nie mogą być współdzielone między żadnymi procesami (w tym także zmienne występujące w głównym programie, który traktujemy jako jeszcze jeden proces, muszą być rozłączne ze zmiennymi, które występują w procesach). Wyjątkiem są zmienne buforowe, które zawsze są globalne. Rozwiązanie to nie zmieni się, jeśli uruchomimy więcej procesów:

```
type
  Komunikaty = (Bilet)
var
  b: buffer;

process P (i: integer);
var
  k: Komunikaty;
begin
  repeat
    sekcja_lokalna;
    GetMessage (b, k)
    rejon_krytyczny;
    SendMessage (b, k)
  until false
```

```
end;

const
  ileP = ...;
var i: integer;
begin
  SendMessage (b, Bilet);
  cobegin for i := 1 to ileP do P(i) coend
end
```

Czasami zachodzi jednak konieczność ograniczenia liczby procesów jednocześnie przebywających w sekcji krytycznej niekoniecznie do jednego procesu, ale ogólnie do pewnej ustalonej wielkości `M` $<$ `ileP`. Wtedy wystarczy umieścić w buforze odpowiednią liczbę „biletów wstępu".

Inne możliwe rozwiązanie polega na wprowadzeniu dodatkowego procesu `Dozorcę`, który będzie nadzorował wykorzystanie sekcji krytycznej. Proces, który chce skorzystać z sekcji krytycznej musi wysłać zamówienie do dozorcy i oczekiwać na potwierdzenie. Do komunikacji wykorzystamy zatem dwa bufory: jeden do wysyłania komunikatów do dozorcy, a drugi do odbierania zezwoleń (dlaczego **muszą** to być dwa oddzielne bufory?):

```
type
  Komunikaty = (Chce, Możesz, Zwalniam)
var
  zam, zezw: buffer;

process P (i: integer);
var
  k: Komunikaty;
begin
  repeat
    sekcja_lokalna;
    SendMessage (zam, Chce);
    GetMessage (zezw, k);
    rejon_krytyczny;
    SendMessage (zam, Zwalniam);
  until false
end;

const
  M = ...;

process Dozorca;
var
  k: Komunikaty;
  ileWolnych: integer := M;
  iluChce: integer := 0;
begin
  repeat
    GetMessage (zam, k);
    case k of
     Chce: if ileWolnych > 0 then
           begin
             dec (ileWolnych);
             SendMessage (zezw, Możesz)
           end
           else
```

```
            inc (iluChce)
     Zwalniam: if iluChce > 0 then
           begin
             SendMessage (zezw, Możesz);
             dec (iluChce)
           end else
             inc (ileWolnych)
    end
  until false
end;

const
  ileP = ...;
var i: integer;
begin
  cobegin Dozorca; for i := 1 to ileP do P(i) coend
end
```

Ten algorytm ma pewną subtelność: procesy nie muszą wchodzić do sekcji krytycznej w kolejności jej zamawiania. Może się zdarzyć, że zezwolenie „przeznaczone" dla jednego procesu odbierze proces, który zgłosił zapotrzebowanie później. Na mocy żywotności operacji pobierania z bufora nie dojdzie jednak do zagłodzenie żadnego procesu.

Obsługa procesów zgodnie z kolejnością zgłoszeń wymagałaby wykorzystania dodatkowych informacji umieszczanych w bilecie.

Problem 2. Producenci i konsumenci Problem producentów i konsumentów ma kilka odmian. W notacji z wykładu najłatwiej zapisać rozwiązanie w wersji z nieograniczonym buforem. Wszelkie komentarze są tu chyba zbyteczne:

```
var
  b: buffer;

process Producent;
var
  p: Porcja;
begin
  repeat
    Produkuj (p);
    SendMessage (b, p)
  until false
end;

process Konsument;
var
  p: Porcja;
begin
  repeat
    GetMessage (b, p);
    Konsumuj (p)
  until false
end;

begin
  cobegin Producent; Konsument coend
end
```

Powyższe rozwiązanie jest także poprawne dla wielu producentów i wielu konsumentów. Trudniej jest uzyskać rozwiązanie w wariancie problemu z buforem o pojemności `M` porcji. Trzeba wtedy „symulować" ograniczoność bufora w sposób podobny do pierwszego rozwiązania problemu wzajemnego wykluczania. W osobnym buforze umieszczamy „bilety", z których każdy reprezentuje jedno wolne miejsce w buforze. Producent przed umieszczeniem porcji w buforze pobiera bilet:

```
type
  Bilety = (Bilet);
var
  b: buffer;
  bpom: buffer;

process Producent;
var
  p: Porcja;
  bil: Bilety;
begin
  repeat
    Produkuj (p);
    GetMessage (bpom, bil);
    SendMessage (b, p)
  until false
end;

process Konsument;
var
  p: Porcja;
begin
  repeat
    GetMessage (b, p);
    SendMessage (bpom, Bilet)
    Konsumuj (p)
  until false
end;

var
  i: 1..M;
begin
  for i := 1 to M do SendMessage (bpom, Bilet);
  cobegin Producent; Konsument coend
end
```

Jeszcze innego rozwiązania wymaga żądanie, żeby producent przekazywał porcję „bezpośrednio" do konsumenta (tzn. że możliwa jest produkcja kolejnej porcji dopiero gdy konsument odbierze dane). Konsument musi wtedy potwierdzać fakt odebrania komunikatu:

```
type
  Potw = (Potwierdzenia);
var
  b: buffer;
  bpom: buffer;

process Producent;
var
  p: Porcja;
```

```
  pot: Potw;
begin
  repeat
    Produkuj (p);
    SendMessage (b, p);
    GetMessage (bpom, pot)
  until false
end;

process Konsument;
var
  p: Porcja;
begin
  repeat
    GetMessage (b, p);
    SendMessage (bpom, Potwierdzenie)
    Konsumuj (p)
  until false
end;

begin
  cobegin Producent; Konsument coend
end
```

Problem 3. Czytelnicy i Pisarze Wydaje się, że rozwiązanie tego problemu można ująć następującym algorytmem. Czytelnik, który chce czytać sprawdza, czy w czytelni nie ma pisarza. Jeśli nie ma, to rozpoczyna czytanie. Pisarz, który chce pisać czeka aż czytelnia będzie pusta i rozpoczyna pisanie. Implementując taki algorytm dobrze jest wprowadzić dodatkowy proces `Czytelnia` synchronizujący dostęp do czytelni. Proces ten przechowuje informacje o tym, kto jest w czytelni i decyduje, kogo można wpuścić. Do komunikacji wprowadzimy trzy bufory: jeden do komunikacji z czytelnią, jeden do wysyłania zezwoleń dla pisarzy, jeden do wysyłania zezwoleń dla czytelników:

```
type
  Komunikaty = (ChceCzytac, ChcePisac, Wychodze);
  Zezwolenia = (Wejdz)

var
  czytelnia, czyt, pis: buffer;

process Czytelnik (i: integer);
var
  z: Zezwolenia;
begin
  repeat
    Sekcja_lokalna;
    SendMessage (czytelnia, ChceCzytac);
    GetMessage (czyt, z);
    CZYTANIE;
    SendMessage (czytelnia, Wychodze)
  until false
end;

process Pisarz (i: integer);
var
```

```
  z: Zezwolenia;
begin
  repeat
    sekcja_lokalna;
    SendMessage (czytelnia, ChcePisac);
    GetMessage (pis, z);
    PISANIE;
    SendMessage (czytelnia, Wychodze)
  until false
end;

process Czytelnia;
var
  dc, dp : integer; { liczba odp. czytelnikow i pisarzy
                      w czytelni. Dla symetrii dp jest
                      typu integer, choć mogłaby być typu
                      boolean }
  ac, ap : integer; { liczba czyt. i pis., którzy się
                      pojawili, więc ac - dc czytelników
                      oczekuje na wejście do czytelni }
  kom: Komunikaty;
begin
  dc := 0; dp := 0;
  repeat
    GetMessage (czytelnia, kom);
    case kom of
     ChceCzytac: begin
                   inc (ac);
(*)                if dp = 0 then
                   begin
                     inc (dc);
                     SendMessage (czyt, Wejdz);
                   end
                 end;
     ChcePisac:  begin
                   inc (ap);
                   if dp + dc = 0 then
                   begin
                     inc (dp);
                     SendMessage (pis, Wejdz)
                   end
                 end
     Wyjdz     : if dp > 0 then { wychodzi pisarz }
                 begin
                   dec (dp); dec (ap);
                   if dc < ac then
                     while dc < ac do
                     begin
                       SendMessage (czyt, Wejdz)
                       inc (dc)
                     end
                   else if dp < ap then begin
                     inc (dp);
                     SendMessage (pis, Wejdz)
```

```
                  end
                end else { wychodzi czytelnik }
                begin
                  dec (dc); dec (ac);
                  if (dc = 0) and (dp < ap) then begin
                    inc (dp);
                    SendMessage (pis, Wejdz)
                  end
                end
    end {case}
  until false
end;
```

Zwróćmy uwagę na sposób obsługi wychodzenia z czytelni. Pisarz, który wychodzi sprawdza najpierw, czy czekają czytelnicy. Jeśli tak, to wpuszcza wszystkich czekających. Dopiero gdy upewni się, że nie ma oczekujących czytelników sprawdza, czy czekają pisarze i jeśli tak, to wpuszcza jednego z nich. Odwrotna kolejność mogłaby spowodować zagłodzenie czytelników w razie ciągłego zgłaszania się pisarzy. Analogicznie czytelnik, który jako ostatni wychodzi z czytelni wpuszcza jednego z oczekujących pisarzy (jeśli są). Czytelnicy na pewno w tym momencie nie czekają (skoro w czytelni byli czytelnicy to nowi czytelnicy mogli od razu wejść do czytelni).

Niestety zachowanie odpowiedniej kolejności wpuszczania do czytelni przy wychodzeniu z niej nie wystarcza do uniknięcia zagłodzenia. W powyższym rozwiązaniu czytelnicy mogą z łatwością zagłodzić pisarzy: wystarczy, że czytelnia będzie wiecznie okupowana przez czytelników. Aby temu zapobiec czytelnia przestaje wpuszczać nowych czytelników, gdy pojawi się oczekujący pisarz. W tym celu warunek z wiersza oznaczonego gwiazdką zmieniamy na `ap= 0`.

Rozwiązanie tego problemu można znacznie uprościć zakładając, że liczba miejsc w czytelni jest ograniczona i wynosi `M`. Wtedy ponownie wykorzystujemy pomysł „biletów" na wejście do sekcji krytycznej: czytelnik przed rozpoczęciem czytania musi pobrać jeden bilet, a pisarz musi uzyskać komplet biletów. Dodatkowy proces jest wtedy niepotrzebny, wystarczy też jeden bufor:

```
type
  Bilety = (Bilet);
var
  czytelnia : buffer;

process Czytelnik (i: integer);
var
  bil: Bilety;
begin
  repeat
    Sekcja_lokalna;
    GetMessage (czytelnia, bil);
    CZYTANIE;
    SendMessage (czytelnia, Bilet)
  until false
end;

process Pisarz (i: integer);
var
  bil: Bilety;
  i: 1..M;
begin
  repeat
    sekcja_lokalna;
```

```
    for i := 1 to M do
      GetMessage (czytelnia, bil);
    PISANIE;
    for i := 1 to M do
      SendMessage (czytelnia, Bilet)
  until false
end;
```

Niestety powyższe rozwiązanie nie jest poprawne. Może dojść do zakleszczenia, jeśli dwóch pisarzy będzie jednocześnie próbować dostać się do czytelni. Wtedy jeden z nich może zabrać część biletów, a drugi pozostałe i obydwaj będą czekać w nieskończoność na pozostałe bilety blokując siebie i inne procesy. Prostym rozwiązaniem tego problemu jest umieszczenie pierwszej pętli pisarza w sekcji krytycznej:

```
type
  Bilety = (Bilet);
var
  czytelnia, pis : buffer;

process Czytelnik (i: integer);
var
  bil: Bilety;
begin
  repeat
    Sekcja_lokalna;
    GetMessage (czytelnia, bil);
    CZYTANIE;
    SendMessage (czytelnia, Bilet)
  until false
end;

process Pisarz (i: integer);
var
  bil: Bilety;
  i: 1..M;
begin
  repeat
    sekcja_lokalna;
    GetMessage (pis, bil);
    for i := 1 to M do
      GetMessage (czytelnia, bil);
    SendMessage (pis, bil);
    PISANIE;
    for i := 1 to M do
      SendMessage (czytelnia, Bilet)
  until false
end;
```

Oczywiście początkowo w buforze `czytelnia` musi być `M` biletów, a w buforze `pis` — jeden bilet.