Część I
Rozwiązania klasycznych problemów w Rendezvous

1 Producenci i konsumenci

Na początek rozważmy wersję z jednym producentem i jednym konsumentem, działającymi w nieskończonych pętlach. Mechanizm komunikacji synchronicznej umożliwia rozwiązanie problemu producenta i konsumenta bez jakiegokolwiek bufora. Producent przekazuje wyprodukowaną porcję bezpośrednio do konsumenta; dopiero po jej odebraniu może wyprodukować następną.

```
type porcja;
void produkuj(porcja &p);
void konsumuj(porcja p);

process Producent {
    porcja p;
    while (true) {
        produkuj(p);
        send Konsument.p;
    }
}

process Konsument {
    porcja p;
    while (true) {
        receive p;
        konsumuj(p);
    }
}
```

Jeśli Producent produkuje tylko skończoną liczbę porcji, nieznaną Konsumentowi, poprawne zakończenie obu procesów wymaga wprowadzenia sygnału koniec().

```
process Konsument {
    porcja p;
    bool dalej = true;
    while (dalej) select {
        on p
            konsumuj(p);
        on koniec()
            dalej = false;
    }
}
```

```
process Producent {
    porcja p;
    for (i in 0..N-1) {
        produkuj(p);
        send Konsument.p;
    }
    send Konsument.koniec();
}
```

Teraz możemy zmodyfikować nasz problem zastępując jednego konsumenta przez K konsumentów. Zastosowanie mechanizmu komunikacji synchronicznej jest trudniejsze niż w przypadku komunikacji asynchronicznej. Producent przed wysłaniem porcji musi wiedzieć, do którego konsumenta może ją przekazać. Jest to niezbędne po to, aby nie próbował wysłać porcji do zajętego konsumenta, podczas gdy inny oczekiwałby na nią. Dlatego wprowadzimy dodatkowy sygnał chcę(), który proces Konsument wysyła wtedy, gdy chce otrzymać kolejną porcję.

```
process Producent {
   porcja p;
   while (true) {
      produkuj(p);
      select
         for (i in 1..K)              // niedeterministyczny
            on Konsument[i].chcę()    // wybór
               send Konsument[i].p;   // gotowego konsumenta
   }
}

process Konsument[i : 1..K] {
    porcja p;
    while (true) {
        send Producent.chcę();
        receive p;
        konsumuj(p);
    }
}

process Producent {          // Wersja 2: przesyłanie pidów
    porcja p;
    while (true) {
        produkuj(p);
        receive chcę(i);
        send Konsument[i].p;
    }
}
```

```
process Konsument[i : 1..K] {
    porcja p;
    while (true) {
        send Producent.chcę(i);
        receive p;
        konsumuj(p);
    }
}
```

Teraz rozważmy wersję problemu z jednym producentem, jednym konsumentem i jednoelementowym buforem. Ponieważ w Rendezvous procesy nie mogą komunikować się przez wspólną pamięć, rozwiązanie z buforem wymaga wprowadzenia dodatkowego procesu Bufor komunikującego się bezpośrednio z procesami Producent i Konsument.

```
process Producent {
    porcja p;
    while (true) {
        produkuj(p);
        send Bufor.p;
    }
}

process Konsument {
    porcja p;
    while (true) {
        receive p;
        konsumuj(p);
    }
}

process Bufor {
    porcja p;
    while (true) {
        receive p;
        send Konsument.p;
    }
}
```

Zastanówmy się, co zmieni się w tym rozwiązaniu, jeśli zamienimy bufor jednoelementowy na M–elementowy cykliczny. Może się wydawać, że treści procesów Producent i Konsument powinny pozostać bez zmian. Tak jednak być nie może, ponieważ proces Bufor musi w jakiś sposób zdecydować, kiedy wykonać instrukcję wyjścia, a kiedy wejścia. Do podjęcia takiej decyzji nie wystarczy sprawdzenie, czy bufor jest niepusty, bowiem wówczas proces Bufor mógłby oczekiwać na spotkanie z procesem Konsument wtedy, gdy tamten jest zajęty

konsumowaniem. W tym samym czasie producent nie mógłby nic włożyć do bufora. Zmniejszyłoby to wykorzystanie bufora i niepotrzebnie wstrzymywało producenta. Dlatego należy wprowadzić sygnał chcę(), który proces Konsument wysyła wtedy, gdy jest gotowy na pobranie nowej porcji z bufora.

```
process Producent {
    porcja p;
    while (true) {
        produkuj(p);
        send Bufor.p;
    }
}

process Konsument {
    porcja p;
    while (true) {
        send Bufor.chcę();
        receive p;
        konsumuj(p);
    }
}

process Bufor {
    porcja bufor[M];
    int pozycja = 0;
    int ile = 0;
    while (true) select {
        if (ile < M)
            on bufor[(pozycja + ile) % M]
                ++ile;
        if (ile > 0)
            on chcę() {
                send Konsument.bufor[pozycja++];
                pozycja %= M;
                --ile;
            }
    }
}
```

Wszystkie poprzednie rozwiązania miały bufor scentralizowany lub nie miały go wcale. Teraz rozważmy bufor rozproszony. Istnieje tutaj kilka wariantów organizacji bufora. Na początek zajmijmy się buforem, w którym kolejność wkładania i wyjmowania nie jest istotna (np. przechowujemy materiały nie ulegające przeterminowaniu).

```
process Producent {
    porcja p;
    while (true) {
        produkuj(p);
        select for (i : 1.. M)
            on Bufor[i].pusty()      // receive pusty(i);
                send Bufor[i].p;     // send Bufor[i].p;
    }
}

process Konsument {
    porcja p;
    while (true) {
        receive p;
        konsumuj(p);
    }
}

process Bufor[i : 1..M] {
    porcja p;
    while (true) {
        send Producent.pusty();
        receive p;
        send Konsument.p;
    }
}
```

Powstaje naturalne pytanie, co zrobić aby porcje były pobierane w kolejności wstawiania. W tym celu możemy inaczej zorganizować pracę bufora, producenta i konsumenta. Producent będzie przekazywać porcje do pierwszego bufora, ten do drugiego, a ostatni bufor do konsumenta. W tej wersji Bufor musi znać swój numer w tablicy.

```
process Producent {
    porcja p;
    while (true) {
        produkuj(p);
        send Bufor[1].p;
    }
}

process Bufor[i : 1..M] {
    porcja p;
    while (true) {
        receive p;
        if (i == M)
            send Konsument.p;
```

```
        else
            send Bufor[i + 1].p;
    }
}

process Konsument {
    porcja p;
    while (true) {
        receive p;
        konsumuj(p);
    }
}
```

Powyższe rozwiązanie ma pewną wadę. Jest nią duży czas przesyłania produktu od producenta do konsumenta. Aby tego uniknąć można wprowadzić przełącznik wskazujący który bufor mamy zapełnić (opróżnić) jako następny.

```
process Producent {
    porcja p;
    int kolejny = 1;
    while (true) {
        produkuj(p);
        send Bufor[kolejny++].p;
        kolejny %= M;
    }
}

process Konsument {
    porcja p;
    int kolejny = 1;
    while (true) {
        receive Bufor[kolejny++].p;
        kolejny %= M;
        konsumuj(p);
    }
}

process Bufor[i : 1..M] {
    porcja p;
    while (true) {
        receive p;
        send Konsument.p;
    }
}
```

2 Czytelnicy i pisarze z zamianą ról

W systemie działa serwer synchronizujący pracę procesów. Każdy proces cyklicznie załatwia własne sprawy, po czym wchodzi do sekcji krytycznej. Dostęp do sekcji może być *wyłączny* lub *dzielony*. Proces żądający *wyłącznego* dostępu do sekcji krytycznej może w niej przebywać jako jedyny. Proces żądający *dzielonego* dostępu może korzystać z sekcji krytycznej w tym samym czasie co inne procesy żądające dostępu dzielonego. Ponado proces będąc w sekcji krytycznej może *co najwyżej raz* zażądać zmiany sposobu korzystania z sekcji. Procesy żądające zmiany sposobu dostępu z *dzielonego* na *wyłączny* zostają wstrzymane do momentu, w którym będą mogły korzystać z sekcji na zasadzie wyłączności. Procesy te mają pierwszeństwo w dostępie do sekcji przed nowymi procesami.

Napisz treść procesu serwera i procesów żądających dostępu do sekcji krytycznej w sposób dzielony oraz wyłączny.

```
#define C 10
#define P 10

void czytanie();
void pisanie();

process Czytelnik[i : 1..C] {
   while (true) {
      send Czytelnia.chcęCzytać(i);
      receive możeszCzytać();
      czytanie;
      if (nie_zmieniam)
         send Czytelnia.koniecCzytania();
      else {
         send Czytelnia.zmiana(i);
         receive możeszPisać();
         pisanie;
         send Czytelnia.koniecPisania();
      }
   }
}

process Pisarz[i : 1..P] {
   while (true) {
      send Czytelnia.chcęPisać();
      receive możeszPisać();
      pisanie();
      if (nie_zmieniam)
         send Czytelnia.koniecPisania();
      else {
         send Czytelnia.zmiana();
```

```
            czytanie;
            send Czytelnia.koniecCzytania();
        }
    }
}

void CzekajNaPusta () {          // tylko gdy są czytelnicy
    int kto;

    if (ile_czyta > 0)  select {

        on koniecCzytania() {     // koniec pracy czytelnika
            ile_czyta--;
            CzekajNaPusta();
        }

        on zmiana(kto) {     // zamiana czytelnika w pisarza
            ile_czyta--;
            CzekajNaPusta();
            send Czytelnik[kto].możeszPisać();
            receive koniecPisania();
        }
    }
}

void CzekajNaPusta () {      // wersja iteracyjna
    int kto, i;
    int ktoCzeka[C];         // kto czeka na zmianę
    int ileCzeka = 0;        // ilu czeka na zmianę

    while (ile_czyta > 0)
        select {
            on koniecCzytania()     // koniec pracy czytelnika
                ile_czyta--;

            on zmiana(kto) {        // zmiana czytelnika w pisarza
                ile_czyta--;
                ktoCzeka[ileCzeka++] = kto;
        }
    for (i = 0; i < ileCzeka; i++) {       // wpuszczanie wg
                                           // kolejności zgłoszeń
        send Czytelnik[ktoCzeka[i]].możeszPisać();
        receive koniecPisania();
    }
    ileCzeka = 0;
}
```

```
process Czytelnia {
int czytający = 0;
int kto;

   while (true) select {

      on chcęCzytać(kto) {
          czytający++;
          send Czytelnik[kto].możeszCzytać();
      }

      on koniecCzytania()
          czytający--;

//    if (czytajacy > 0)
      on zmiana(kto) {          // zmiana czytelnika w pisarza
          czytajacy--;
          CzekajNaPusta();
          send Czytelnik[kto].możeszPisać();
          receive koniecPisania();
      }

      on chcęPisać(kto) {
          CzekajNaPusta();
          send Pisarz[kto].możeszPisać();
          select {
             on koniecPisania();
             on Pisarz[kto].zmiana()          // zmiana pisarza
                 czytający++;                 // ... = 1
          }
      }
   }
}
```

3 Czytelnicy i pisarze

Mamy do dyspozycji czytelnię posiadającą nieograniczoną liczbę miejsc. Z czytelni chcą korzystać czytelnicy i pisarze. W czytelni może przebywać wielu czytelników naraz, jednakże pisarz potrzebuje skupienia i kiedy pisze, to obecność jakiejkolwiek innej osoby rozprasza go.

Chcemy napisać rozwiązanie, które umożliwiałoby wchodzenie nowych czytelników do czytelników już przebywających w czytelni. Zezwalamy tym samym na zagłodzenie pisarzy. Zakładamy, że mamy C czytelników i P pisarzy.

```
#define C 10
#define P 10

void czytanie();
void pisanie();

process Czytelnik[C] {
    while (true) {
        send Czytelnia.czytam();
        czytanie();
        send Czytelnia.koniecCzytania();
    }
}

process Pisarz[P] {
    while (true) {
        send Czytelnia.piszę();
        pisanie();
        send Czytelnia.koniecPisania();
    }
}

process Czytelnia {
    int czytający = 0;
    while (true) select {
        on czytam()
            ++czytający;
        on koniecCzytania()
            --czytający;
        if (czytający == 0)
            on piszę()
                receive koniecPisania();
    }
}
```

Teraz napiszmy rozwiązanie poprawne, które uzyskujemy w łatwy sposób z poprzedniego przez dołożenie sygnału chcęPisać() do procesu Pisarz.

```
#define C 10
#define P 10

void czytanie();
void pisanie();

process Czytelnik[i : 1..C] {
    while (true) {
        send Czytelnia.czytam();
        czytanie();
        send Czytelnia.koniecCzytania();
    }
}

process Pisarz[i : 1..P] {
    while (true) {
        send Czytelnia.chcęPisać();
        receive możeszPisać();
        pisanie();
        send Czytelnia.koniecPisania();
    }
}

process Czytelnia {
    int czytający = 0;
    while (true) select {
        on czytam()
            ++czytający;
        on koniecCzytania()
            --czytający;
        for (i : 1..P)
            on Pisarz[i].chcęPisać() {
                while (czytający > 0) {
                    receive koniecCzytania();
                    --czytający;
                }
                send Pisarz[i].możeszPisać();
                receive koniecPisania();
            }
    }
}
```

Subtelność tego rozwiązania polega na wykorzystaniu żywotności niedeterminizmu występującego w Rendezvous. Jeśli na wejście do czytelni czekają zarówno pisarze jak i czytelnicy, to zostanie wpuszczony któryś z tych procesów, ale nie mamy wpływu na to który. Rozwiązanie nie doprowadzi do zagłodzenia

żadnej grupy dzięki żywotności dozorów. Jeśli chcielibyśmy sami decydować o tym, kogo wpuszczamy do czytelni i, na przykład, po wyjściu pisarza wpuszczać wszystkich oczekujących czytelników, aby w pełni wykorzystać czytelnię, to rozwiązanie skomplikuje się nieco.

```
#define C 10
#define P 10

void czytanie();
void pisanie();

process Czytelnik[i : 1..C] {
    while (true) {
        send Czytelnia.chcęCzytać();
        send Czytelnia.zaczynamCzytanie();
        czytanie();
        send Czytelnia.koniecCzytania();
    }
}

process Pisarz[i : 1..P] {
    while (true) {
        send Czytelnia.chcęPisać();
        send Czytelnia.zaczynamPisanie();
        pisanie();
        send Czytelnia.koniecPisania();
    }
}

process Czytelnia {
    int czytający = 0;
    int czekającyCzytelnicy = 0;
    int piszący = 0;
    int czekającyPisarze = 0;
    while (true) select {
        on chcęCzytać()
            if (piszący + czekającyPisarze == 0) {
                ++czytający;
                receive zaczynamCzytanie();
            } else
                ++czekającyCzytelnicy;
        on koniecCzytania()
            if (--czytający == 0 && czekającyPisarze > 0) {
                receive zaczynamPisanie();
                piszący = 1;
                --czekającyPisarze;
```

```
            }
        on chcęPisać()
            if (czytający + piszący == 0) {
                piszący = 1;
                receive zaczynamPisanie();
            } else
                ++czekającyPisarze;
        on koniecPisania() {
            piszący = 0;
            if (czekającyCzytelnicy > 0) {
                while (czekającyCzytelnicy > 0) {
                    receive zaczynamCzytanie();
                    --czekającyCzytelnicy;
                }
            } else if (czekającyPisarze > 0) {
                receive zaczynamPisanie();
                piszący = 1;
                --czekającyPisarze;
            }
        }
    }
}
```