Część III
Zakleszczenie

1 Wymiana grupowa

W systemie działa `N`-elementowa tablica procesów `Pracownik` wymieniających między sobą wartości typu `porcja`. Porcje są jednego z `R` rodzajów, reprezentowanych przez nieujemne liczby całkowite. Rodzaj porcji można sprawdzić funkcją

```
int rodzaj(porcja p);
```

Pracownik, w pętli, produkuje porcję funkcją

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

a następnie zdobywa porcję tego samego rodzaju, wyprodukowaną przez innego pracownika, i konsumuje ją funkcją

```
void konsumuj(porcja p);
```

Zakładamy, że każdy pracownik produkuje porcję każdego rodzaju nieskończenie wiele razy. Porcje między pracownikami są przekazywane przez procesy `N`-elementowej tablicy `Pomocnik`. Proces `Pomocnik[i]` może wysyłać komunikaty tylko do `Pracownik[i]` oraz `Pomocnik[(i + 1) % N]`.

Zaimplementuj procesy `Pomocnik` zakładając, że pracownicy wykonują algorytm:

```
process Pracownik[id : 1..N] {
    while (true) {
        porcja p;
        produkuj(p);
        send Pomocnik[id].wymień(p);
        receive p;
        konsumuj(p);
    }
}
```

Rozwiązanie z blokadą

Poprawne rozwiązanie nie powinno dopuszczać do blokady procesów. Krótkie i proste rozwiązanie, przedstawione poniżej, dopuszcza możliwość blokady w sytuacji, gdy wszystkie procesy `Pracownik[i]` równocześnie prześlą porcje do swoich procesów `Pomocnik[i]`. Każdy proces `Pomocnik[i]` będzie wówczas czekał na spotkanie ze swoim następnikiem.

```
#define BRAK (-1)

process Pomocnik[id : 1..N] {
    porcja nowaPorcja;
    int oczekiwany = BRAK;
    int następny = id % N + 1;
    while (true) select {
        on wymień(nowaPorcja) {
            send Pomocnik[następny].przekaż(nowaPorcja, id);
            oczekiwany = rodzaj(nowaPorcja);
        }
        on przekaż(nowaPorcja, nowyAutor)
            if (oczekiwany == rodzaj(nowaPorcja) && nowyAutor != id) {
                send Pracownik[id].nowaPorcja;
                oczekiwany = BRAK;
            } else
                send Pomocnik[następny].przekaż(nowaPorcja, nowyAutor);
    }
}
```

Rozwiązanie z przekazywaniem żetonu

Poprawne rozwiązanie problemu wymaga wprowadzenia uprawnień do przekazania porcji następnikowi (dla procesów `Pomocnik[i]`). Posiadanie uprawnienia do wykonania tej operacji będziemy oznaczać poprzez posiadanie żetonu. W poprawnym rozwiązaniu liczba wprowadzonych żetonów musi być równa co najmniej jeden, by co najmniej jeden proces miał prawo przesyłać, i musi być mniejsza od `N`, by w każdej chwili co najmniej jeden proces nie miał prawa przesyłać porcji.

W przedstawionym rozwiązaniu użyto `N / 2` żetonów — co drugi proces ma żeton, czyli prawo przesyłania, a pozostałe procesy czekają na odbiór komunikatu. Przesłanie komunikatu oznacza równocześnie przekazanie żetonu, czyli uprawnienia (procesy zamieniają się rolami).

Zapewnienie poprawnej pracy systemu wymaga, aby w sytuacji, gdy proces posiadający uprawnienia nie ma niczego do przesłania, nastąpiło przekazanie uprawnienia, czyli przesłanie żetonu (np. sąsiad nie mający uprawnienia otrzymał porcję i czeka na prawo jej przesłania).

Posiadanie żetonu oznacza zatem nie tylko prawo, ale wręcz obowiązek przesłania. Dlatego w przedstawionym rozwiązaniu przyjęto, że proces posiadający żeton nie odbiera żadnego komunikatu, tylko go wysyła. Odebranie porcji może nastąpić dopiero w następnej fazie, czyli w okresie gdy proces nie ma żetonu.

Może się zdarzyć, że proces `Pomocnik[i]` będzie miał dwie porcje: jedną otrzymaną od `Pracownik[i]`, której nie przesłał, gdyż nie miał żetonu, a drugą otrzymaną od swego poprzednika. W takiej sytuacji, po otrzymaniu żetonu, najpierw przesyła swoją porcję, a w następnej kolejności porcję od poprzednika.

```
void obsłużŻeton(int następny, porcja mojaPorcja, int mójAutor) {
    if (mójAutor != BRAK) {
        send Pomocnik[następny].przekaż(mojaPorcja, mójAutor);
        mójAutor = BRAK;
    } else
        send Pomocnik[następny].żeton();
}

process Pomocnik[id : 1..N] {
    porcja mojaPorcja, nowaPorcja;
    int mójAutor = BRAK, nowyAutor;
    int oczekiwany = BRAK;
    int następny = id % N + 1;
    if (id % 2 == 0)
        send Pomocnik[następny].żeton();
    while (true) select {
        on wymień(mojaPorcja) {
            oczekiwany = rodzaj(mojaPorcja);
            mójAutor = id;
        }
        on przekaż(nowaPorcja, nowyAutor)
            if (oczekiwany == rodzaj(nowaPorcja) && nowyAutor != id) {
                send Pracownik[id].nowaPorcja;
                oczekiwany = BRAK;
                obsłużŻeton(następny, mojaPorcja, mójAutor);
            } else if (mójAutor != BRAK) {
                send Pomocnik[następny].przekaż(mojaPorcja, mójAutor);
                mojaPorcja = nowaPorcja;
                mójAutor = nowyAutor;
            } else
                send Pomocnik[następny].przekaż(nowaPorcja, nowyAutor);
        on żeton()
            obsłużŻeton(następny, mojaPorcja, mójAutor);
    }
}
```

2 Zamieniacze

W systemie, w nieskończonych pętlach, działają cztery `N`-elementowe grupy procesów: `Producenci`, `Konsumenci`, `Bufory` i `Zamieniacze`. Każdy `Producent[i]` produkuje porcje funkcją

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

i przekazuje do procesu `Bufor[i]`. Proces `Bufor[i]` przechowuje maksymalnie `K` porcji. Proszony o porcję daje ostatnią otrzymaną, spośród tych, które w danej chwili ma. Proces `Konsument[i]` odbiera porcję od `Bufor[i]` i konsumuje ją funkcją

```
void konsumuj(porcja p);
```

Procesy `Zamieniacz[i]` wykonują

```
void własneSprawy();
```

a następnie, jeśli `Bufor[i]` oraz `Bufor[(i + 1) % N]` nie są puste, zamieniają miejscami po jednej porcji z tych buforów. Zaimplementuj wszystkie wymienione procesy.

Rozwiązanie

```
#define N 100
#define K 10

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

process Producent[id : 1..N] {
    porcja d;
    while (true) {
        produkuj(d);
        send Bufor[id].wstaw(d);
    }
}

process Konsument[id : 1..N] {
    porcja d;
    while (true) {
        send Bufor[id].daj();
        receive d;
        konsumuj(d);
    }
}

process Zamieniacz[id : 1..N] {
    porcja d1, d2;
    int najpierw, później;
    if (id < N) {
        najpierw = id;
        później = id + 1;
    } else {
        najpierw = 1;
        później = N;
    }
    while (true) {
        własneSprawy();
        send Bufor[najpierw].dajJeśliMasz(id);
        select {
            on d1 {
                send Bufor[później].dajJeśliMasz(id);
                select {
                    on d2 {
                        send Bufor[najpierw].d2;
                        send Bufor[później].d1;
                    }
```

```
                    on nieMam()
                        send Bufor[najpierw].d1;
                }
            }
            on nieMam()
                ;
        }
    }
}

process Bufor[id : 1..N] is
    porcja przechowywane[K];
    int ile = 0, komu;
    while (true) select {
        if (ile < K) on wstaw(przechowywane[ile++])
            ;
        if (ile > 0) on daj()
            send Konsument[i].przechowywane[--ile];
        on dajJeśliMasz(komu)
            if (ile == 0) {
                send Zamieniacz[komu].nieMam();
            } else {
                send Zamieniacz[komu].przechowywane[ile - 1];
                receive przechowywane[ile - 1];
            }
    }
}
```

3 Kopiec

Poniższy fragment kodu implementuje sekwencyjnie, za pomocą kopca binarnego, kolejkę priorytetową przechowującą maksymalnie `N` liczb całkowitych.

```
#define K 200
#define N 100

void wstaw(int x) {
    send Q[1].x;
}

int usuńMin(int id) {
    int x;
    send Q[1].chcęUsunąć(id);
    receive x;
    return x;
}

process P[id : 1..K] {
    int x;
    ...
    wstaw(x);
    ...
    x = usuńMin(id);
    ...
}

process Q[1] {
    int wolna = 1;
    int t[N + 1];
    int kto, i, x;
    while (true) select {
        if (wolna <= N) on x {
            int ojciec, v;
            i = wolna++;
            while ((ojciec = i / 2) > 0 && (v = t[ojciec]) > x) {
                t[i] = v;
                i = ojciec;
            }
```

```
            t[i] = x;
        }
        if (wolna > 1) on chcęUsunąć(kto) {
            int syn, v, w;
            int r = t[1];
            if (--wolna > 1) {
                x = t[wolna];
                for (i = 1; (syn = 2 * i) < wolna; i = syn) {
                    v = t[syn];
                    if (syn + 1 < wolna && (w = t[syn + 1]) < v) {
                        ++syn;
                        v = w;
                    }
                    if (v >= x)
                        break;
                    t[i] = v;
                }
                t[i] = x;
            }
            send P[kto].r;
        }
    }
}
```

Zaimplementuj w Rendezvous rozproszony wariant tej struktury danych. Kod korzystających z kolejki procesów `P` oraz funkcji

```
void wstaw(int x);
int usuńMin(int i);
```

powinien pozostać bez zmian. Wartości w kolejce mają być przechowywane, po jednej, przez procesy odpowiednio powiększonej tablicy procesów `Q`. Zadbaj o poprawność i jak największą współbieżność rozwiązania.

Rozwiązanie

```
void wstaw(int x) {
    send Q[0].x;
}

int usuńMin(int i) {
    int x;
    send Q[0].chcęUsunąć(i);
    receive x;
    return x;
}

process P[id : 1..K] {
    int x;
    ...
    wstaw(x);
    ...
    x = usuńMin(i);
    ...
}

process Q[id : 0..N] {
    if (id == 0) {
        int wstawiane = 0, usuwane = 0, ile = 0, x, kto;
        while (true) select {
            if (ile > 0 && wstawiane == 0) on chcęUsunąć(kto) {
                send Q[1].daj(0);
                receive x;
                send P[kto].x;
                if (--ile > 0) {
                    send Q[ile].daj(0);
                    receive x;
```

```
                send Q[1].wDół(x);
                ++usuwane;
            }
        }
        if (ile < N && usuwane == 0) on x {
            send Q[++ile].wGórę(x);
            ++wstawiane;
        }
        on wstawione()
            --wstawiane;
        on usunięte()
            --usuwane;
    }
} else {
    int x, moja, minimum, czyja, kto;
    while (true) {
        select {
            on daj(kto) {
                send Q[kto].nieMam();
                continue;
            }
            on rezygnuję()
                ;
            on wGórę(moja)
                if (i == 1) {
                    send Q[0].wstawione();
                } else {
                    send Q[i / 2].daj(i);
                    receive x;
                    if (x > moja) {
                        send Q[i / 2].wGórę(moja);
                        moja = x;
                    } else {
                        send Q[i / 2].rezygnuję();
                        send Q[0].wstawione();
                    }
                }
            on wDół(moja) {
                minimum = moja;
                czyja = i;
                for (int j = 2 * i; j <= 2 * i + 1 && j <= N; ++j) {
                    send Q[j].daj(i);
                    select {
                        on nieMam()
                            break;
                        on x
                            if (x < minimum) {
                                if (czyja != i)
                                    send Q[czyja].rezygnuję();
                                minimum = x;
                                czyja = j;
                            } else
                                send Q[j].rezygnuję();
                    }
                }
                if (czyja == i) {
                    send Q[0].usunięte();
                } else {
                    send Q[czyja].wDół(moja);
                    moja = minimum;
                }
            }
        }
        receive daj(kto);
        send Q[kto].moja;
    }
}
}
```