Programowanie współbieżne

Ćwiczenia 1 – problem wzajemnego wykluczania

Zadanie 1

Uruchamiamy współbieżnie dwa następujące procesy.

```
int y = 0;

process P1() {
  int x, i;

  for (i = 1; i < 6; i++) {
    x = y;
    x = x + 1;
    y = x;
  }
}
```

```
process P2() {
  int x, i;

  for (i = 1; i < 6; i++) {
    x = y;
    x = x + 1;
    y = x;
  }
}
```

Jaką wartość będzie miała zmienna y po zakończeniu działania obu procesów? Przyjmujemy, że zmienna zadeklarowana poza treścią procesu jest zmienną globalną, do której dostęp mają wszystkie procesy.

Zadanie 2

Uruchamiamy współbieżnie dwa następujące procesy.

```
process P1() {
  while (true) {
    sekcja_lokalna;
    protokol_wstepny;
    sekcja_krytyczna;
    protokol_koncowy
  }
}
```

```
process P2() {
  while (true) {
    sekcja_lokalna;
    protokol_wstepny;
    sekcja_krytyczna;
    protokol_koncowy
  }
}
```

Chcemy zapewnić, że w tym samym czasie co najwyżej jeden z nich wykonuje fragment programu oznaczony jako *sekcja_krytyczna*. Jakie instrukcje należy umieścić w protokołach, aby zrealizować ten cel? Zakładamy, że nie dysponujemy żadnymi mechanizmami synchronizacyjnymi, więc protokoły powinny umiejętnie wykorzystać zmienne globalne oraz instrukcje języka programowania.

- Czy protokoły wstępne i końcowe można pozostawić puste?
- Co oznacza bezpieczeństwo w przypadku tak sformułowanego zadania?
- Co oznacza żywotność?

W rozwiązaniu będziemy korzystać ze zmiennych globalnych i lokalnych. *Zmienna lokalna* może znajdować się w prywatnej przestrzeni adresowej procesu. Pozostałe procesy nie mają do niej dostępu, nie mogą jej zatem ani odczytywać ani modyfikować. Inaczej sytuacja wygląda ze *zmiennymi globalnymi*. Są one współdzielone między procesami, co oznacza, że w dowolnej chwili każdy z nich może takie zmienne zmodyfikować lub je odczytywać.

- Co dzieje się, gdy dwa wykonujące się równolegle procesy w tej samej chwili chcą uzyskać dostęp do tej samej zmiennej, a zatem do tej samej komórki (tych samych komórek) pamięci?

Konflikt rozwiązuje sprzęt za pomocą arbitra pamięci. Jest to układ sprzętowy, który realizuje wzajemne wykluczanie przy dostępie do pojedynczych komórek pamięci. Jednoczesne odwołania do tej samej komórki pamięci zostaną w jakiś nieznany z góry sposób uporządkowane w czasie i wykonane. W dalszej części rozważań zakładamy istnienie arbitra pamięci i jego poprawne działanie.

Pierwsza próba rozwiązania

Spróbujmy najpierw rozwiązać problem wprowadzając zmienną globalną *ktoczeka*. Będzie ona przyjmować wartości 1 lub 2. Wartość 1 oznacza, że proces pierwszy musi poczekać a prawo wejścia do sekcji krytycznej ma proces drugi.

Oczekiwanie na wejście realizujemy poprzez pustą pętlę, której jedynym zadaniem jest cykliczne sprawdzanie warunku na wejście do sekcji krytycznej. Proces, który skorzysta z sekcji krytycznej, przekazuje pierwszeństwo rywalowi.

```
int ktoczeka  = 1;

process P1() {
  while (true) {
    sekcja_lokalna;
    while (ktoczeka == 1) { }
    sekcja_krytyczna;
    ktoczeka = 1;
  }
}
```

```
process P2() {
  while (true) {
    sekcja_lokalna;
    while (ktoczeka == 2) { }
    sekcja_krytyczna;
    ktoczeka = 2;
  }
}
```

Zanim przystąpimy do analizy poprawności rozwiązania, przypomnijmy założenia dotyczące sekcji krytycznej, sekcji lokalnej i systemu operacyjnego, który nadzoruje wykonywanie procesów. Rozstrzygnij, które stwierdzenia są prawdziwe:

- Formułując problem wzajemnego wykluczania zakłada się, że:
  - Każdy proces, który wszedł do sekcji krytycznej w skończonym czasie ją opuści.
  - Proces może przebywać w sekcji krytycznej nie dłużej niż ustalony z góry czas.
  - Proces nie może zakończyć się w sekcji krytycznej.
  - Proces może się zakończyć w sekcji krytycznej ale tylko w sposób poprawny.
- Proces, który rozpoczął wykonywanie sekcji lokalnej:
  - musi w skończonym czasie zakończyć ją i przejść do protokołu wstępnego.
  - może zakończyć swoje działanie na skutek błędu.
  - może zakończyć swoje działanie, ale tylko w sposób poprawny.

Czy wobec przyjętych założeń zaproponowane rozwiązanie poprawne? Własność bezpieczeństwa jest zachowana — nigdy oba procesy nie będą jednocześnie w sekcji krytycznej. A co z własnością żywotności? Przypomnijmy o założeniu, że proces nie przebywa nieskończenie długo w rejonie krytycznym. Zatem po wyjściu z niego (które na pewno w końcu nastąpi) rozpocznie się wykonanie sekcji lokalnej. I tu pojawia się problem, bo o tym fragmencie programu nic założyć nie możemy. Jeśli proces utknie w tym fragmencie kodu (bo nastąpi błąd, zapętlenie, itp.) to drugi z procesów będzie mógł wejść do sekcji krytycznej jeszcze co najwyżej raz. Kolejna próba zakończy się wstrzymaniem procesu na „zawsze". Zatem przedstawione rozwiązanie nie ma własności żywotności. Inną wadą tego rozwiązania jest zbyt ścisłe powiązanie ze sobą procesów. Muszą one korzystać z sekcji krytycznej naprzemiennie, a przecież potrzeby obu procesów mogą być różne. Jeśli ponadto działają one z różną „szybkością" (bo na przykład sekcja lokalna jednego z nich jest bardziej złożona od sekcji lokalnej drugiego), to proces „szybszy" będzie równał tempo pracy do wolniejszego.

Druga próba rozwiązania

Spróbujmy podejść do problemu w inny sposób. Wprowadźmy dwie logiczne zmienne globalne: *jest1* oznaczającą, że proces P1 jest w sekcji krytycznej i analogiczną zmienną *jest2* dla procesu P2. Przed wejściem do sekcji krytycznej proces sprawdza, czy jego rywal jest już w sekcji krytycznej. Jeśli tak, to czeka. Gdy sekcja krytyczna się zwolni to proces ustawi swoją zmienną na *true* sygnalizując, że jest w sekcji krytycznej, po czym wejdzie do niego.

```
bool jest1 = false;
bool jest2 = false;

process P1() {
  while (true) {
    sekcja_lokalna;
    while (jest2) { }
    jest1 = true;
    sekcja_krytyczna;
    jest1 = false;
  }
}
```

```
process P2() {
  while (true) {
    sekcja_lokalna;
    while (jest1) { }
    jest2 = true;
    sekcja_krytyczna;
    jest2 = false;
  }
}
```

Zauważmy, że to rozwiązanie nie uzależnia już procesów od siebie. Jeśli jeden z nich nie chce korzystać z sekcji krytycznej lub awaryjnie zakończy swoje działanie w sekcji lokalnej, to drugi może swobodnie wchodzić do sekcji krytycznej ile razy zechce. Nie ma też problemu z żywotnością. Jeśli pierwszy proces utknął w pętli w protokole wstępnym, to drugi proces musi znajdować się gdzieś między przypisaniem *jest2* na *true* a przypisaniem *jest2* na *false*. Po skończonym czasie wyjdzie zatem z rejonu krytycznego i ustawi zmienną *jest2* na *false* pozwalając partnerowi wyjść z jałowej pętli i wejść do rejonu krytycznego. Niestety, przy pewnych złośliwych przeplotach może się zdarzyć, że do rejonu krytycznego wejdą oba procesy. Wskaż taki scenariusz.

Przyczyną takiej sytuacji jest zbyt późne ustawienie zmiennych logicznych. Proces już „prawie" jest w sekcji krytycznej (bo przeszedł przez wstrzymującą go pętlę), a jeszcze nie poinformował o tym rywala (to jest nie ustawił swojej zmiennej *jest*).

Ponieważ istnieje przeplot, który prowadzi do błędnej sytuacji, więc zgodnie z definicją poprawności (musi być „dobrze" dla każdego przeplotu) stwierdzamy, że powyższy program jest niepoprawny. Zauważmy jednak, że gdyby sprawdzenie warunku w pętli oraz zmiana wartości zmiennej były niepodzielne, to takiego złego scenariusza nie dałoby się pokazać. Tyle tylko, że zagwarantowanie niepodzielności oznacza stworzenie sekcji krytycznej, a to jest właśnie problem, który rozwiązujemy.

Trzecia próba rozwiązania

Ponieważ zmiana wartości zmiennych globalnych w poprzednim rozwiązaniu została dokonana zbyt późno, więc spróbujmy zmienić kolejność instrukcji i ustawmy najpierw zmienne logiczne, a potem dopiero próbujmy przejść przez pętle. Teraz zmienne logiczne oznaczają chęć wejścia do sekcji krytycznej.

```
bool chce1 = false;
bool chce2 = false;

process P1() {
  while (true) {
    sekcja_lokalna;
    chce1 = true;
    while (chce2) { }
    sekcja_krytyczna;
    chce1 = false;
  }
}
```

```
process P2() {
  while (true) {
    sekcja_lokalna;
    chce2 = true;
    while (chce1) { }
    sekcja_krytyczna;
    chce2 = false;
  }
}
```

Teraz mamy program bezpieczny. Faktycznie w rejonie krytycznym może znajdować się co najwyżej jeden proces. Ale brakuje żywotności! Z łatwością można doprowadzić do zakleszczenia. Wskaż odpowiedni scenariusz.

Czwarta próba rozwiązania

Można próbować ratować sytuację zmuszając procesy do chwilowej rezygnacji z wejścia do sekcji i ustąpienia pierwszeństwa.

```
bool chce1 = false;
bool chce2 = false;

process P1() {
  while (true) {
    sekcja_lokalna;
    chce1 = true;
    while (chce2) {
      chce1 = false;
      chce1 = true;
    }
    sekcja_krytyczna;
    chce1 = false;
  }
}
```

```
process P2() {
  while (true) {
    sekcja_lokalna;
    chce2 = true;
    while (chce1) {
      chce2 = false;
      chce2 = true;
    }
    sekcja_krytyczna;
    chce2 = false;
  }
}
```

Niestety znów istnieje złośliwy przeplot, który powoduje brak żywotności. Nie pomoże argumentacja, że taki przeplot jest w zasadzie nieprawdopodobny. Zgodnie z definicją poprawności, skoro istnieje scenariusz powodujący brak żywotności, to program jest niepoprawny.

Algorytm Petersona

Poprawne rozwiązanie znane pod nazwą algorytmu Petersona jest połączeniem pierwszego pomysłu z przedostatnim. Utrzymujemy zmienne *chce1* i *chce2*, które oznaczają chęć wejścia procesu do sekcji krytycznej. Jeśli obydwa procesy chcą wejść do sekcji krytycznej rozstrzygamy konflikt za pomocą zmiennej *ktoczeka*.

```
bool chce1 = false;
bool chce2 = false;
int ktoczeka = 1;

process P1() {
  while (true) {
    sekcja_lokalna;
    chce1 = true;
    ktoczeka = 1;
    while (chce2 && (ktoczeka == 1)) { }
    sekcja_krytyczna;
    chce1 = false;
  }
}
```

```
process P2() {
  while (true) {
    sekcja_lokalna;
    chce2 = true;
    ktoczeka = 2;
    while (chce1 && (ktoczeka == 2)) { }
    sekcja_krytyczna;
    chce2 = false;
  }
}
```

Przypomnijmy, że nie zakładamy atomowości (niepodzielności) poszczególnych instrukcji (w szczególności wyliczanie złożonych warunków logicznych nie musi odbywać się atomowo). Zakładamy natomiast istnienie arbitra pamięci (może dojść do jednoczesnej próby zmiany wartości zmiennej *ktoczeka*).

Zauważmy, że w sytuacji, gdy tylko jeden proces rywalizuje o dostęp do sekcji krytycznej, to będzie on mógł do woli z niej korzystać, bo zmienna *chce* jego rywala ma cały czas wartość *false*. Dzięki temu unikamy ścisłego powiązania procesów, jak było to w rozwiązaniu pierwszym. Z drugiej strony, gdy oba procesy ciągle chcą korzystać z sekcji krytycznej, robią to na zmianę.

Analiza algorytmu

Przeanalizuj algorytm Petersona pod kątem zmiany kolejności wykonywanych instrukcji:

- Czy można zamienić kolejność przypisań przed pętlą while?
- Czy można umieścić zmianę wartości zmiennej *ktoczeka* po wyjściu z sekcji krytycznej?
- Czy można zamienić kolejność sprawdzania warunków w pętli while?
- Czy można zainicjować zmienną *ktoczeka* na inne wartości?
- A zmienne *chce1* i *chce2*?

Wady algorytmu Petersona:

- Aktywne oczekiwanie. Z założenia nie było dostępnych żadnych mechanizmów wstrzymywania procesów, nie pozostało nic innego niż zatrzymanie procesu w jałowej pętli. Angażuje to jednak czas procesora (i szynę danych). W przyszłości aktywne oczekiwanie będziemy traktować jak poważny błąd.
- Liczba procesów musi być znana z góry.
- Koszt protokołu wstępnego jest znaczny.