Programowanie współbieżne

Ćwiczenia 5 – monitory cz. 1

Zadanie 1: Stolik dwuosobowy raz jeszcze

W systemie działa N par procesów. Procesy z pary są nierozróżnialne. Każdy proces cyklicznie wykonuje *własnesprawy*, a potem spotyka się z drugim procesem z pary w kawiarni przy stoliku dwuosobowym (funkcja *randka*()). W kawiarni jest tylko jeden stolik. Procesy zajmują stolik tylko wtedy, gdy obydwa są gotowe do spotkania, natomiast mogą odchodzić od niego pojedynczo, czyli w różnym czasie kończyć wykonywanie funkcji *randka*().

Rozwiązanie

Schemat rozwiązania jest taki sam jak poprzednio. Zamiast semaforów użyjemy następujących zmiennych warunkowych (kolejek):

- zmiennej, na której czekają procesy, które przyszły jako pierwsze z pary,
- zmiennej, na którym drugie procesy z pary czekają na stolik.

Potrzebna będzie również informacja o liczbie procesów siedzących przy stoliku.

```
void randka();

process P(int j) {              /* j to numer pary */
  while (true) {
    własnesprawy;
    Kawiarnia.Chcę_stolik(j);
    randka();
    Kawiarnia.Zwalniam_stolik();
  }
}

monitor Kawiarnia{

  condition naparę[N];
  condition nastół;
  int przystole = 0;

  void Chcę_stolik(int j) {

    if (empty(naparę[j])) wait(naparę[j]);  /* pierwszy z pary */
    else {                                  /* drugi z pary */
      if (przystole > 0) wait(nastół);
      przystole = 2;
      signal(naparę[j]);
    }
  }

  void Zwalniam_stolik() {

    przystole--;
    if (przystole == 0) signal(nastół);
  }
}
```

Zwróćmy uwagę na następujące różnice między zaproponowanym rozwiązaniem a rozwiązaniem korzystającym z semaforów z ćwiczeń 3:

- Nie trzeba synchronizować dostępu procesów do zmiennych monitora (brak jawnego odpowiednika semafora ochrona). Nie musimy tego robić, ponieważ procedur monitora nie może wykonywać wiele procesów jednocześnie.
- Nie potrzebujemy również dodatkowej tablicy przechowującej dla każdej pary informację o tym, czy pierwszy proces z pary już czeka. Wystarczy sprawdzić niepustość kolejki, na której ma czekać ten proces.
- Operacja *wait*() jest zawsze blokująca w przeciwieństwie do semaforowej operacji *P*(), która zależy od wartości semafora. Dlatego w rozwiązaniu korzystającym z semaforów mamy: P(nastół), a w obecnym rozwiązaniu: *if (przystole > 0) wait*(*nastół*).
- Operacja *signal*() na pustej kolejce nie robi nic w przeciwieństwie do operacji *V*(), która wykonana na semaforze, na którym żaden proces nie czeka, powoduje jego podniesienie (otwarcie).

Zadanie 2: Symulacja monitora za pomocą semaforów

Korzystając z semaforów należy napisać odpowiedniki operacji monitorowych:

1. rozpoczęcia wykonywania procedury monitora,
2. operacji *wait*(),
3. operacji *signal*(),
4. zakończenia wykonywania procedury monitora.

Rozwiązanie - wersja 1

W pierwszej wersji rozwiązania zakładamy, że *signal*() jest ostatnią instrukcją w procedurze monitora.

semaphore Monitor = 1; /* *zapewnia wzajemne wykluczanie przy wykonywaniu procedur monitora* */

semaphore Kolejka = 0; /* *każdą zmienną warunkową monitora* */
int ileczeka = 0; /* *symulujemy za pomocą semafora i licznika oczekujących* */

1. **P**(Monitor);

2. ileczeka++;
   V(Monitor);
   P(Kolejka); /* *dziedziczenie sekcji krytycznej* */
   ileczeka−−;

3. **if** (ileczeka > 0) **V**(Kolejka)
   else **V**(Monitor);

4. **V**(Monitor);

Dziedziczenie sekcji krytycznej jest konieczne, aby zachować semantykę operacji monitorowych, czyli zagwarantować, że po wykonaniu *signal*() wykonuje się zwolniony proces. Dokładną symulacją zapewnią wyłącznie semafory silne.

Rozwiązanie - wersja 2

Tym razem nie zakładamy, że *signal*() jest ostatnią instrukcją w procedurze monitora.

```
semaphore Monitor = 1;      /* zapewnia wzajemne wykluczanie
                               przy wykonywaniu procedur monitora */

semaphore Stos = 0;         /* wstrzymuje procesy wstrzymane po signal */
int ileczekas = 0;

semaphore Kolejka = 0;      /* każdą zmienną warunkową monitora */
int ileczekak = 0;          /* symulujemy za pomocą semafora i licznika oczekujących */
```

1. **P**(Monitor);

2. ileczekak++;
   if (ileczekas > 0) **V**(Stos)
   else **V**(Monitor);
   P(Kolejka);
   ileczekak−−;

3. **if** (ileczekak > 0) {
   ileczekas++;
   V(Kolejka);
   P(Stos);
   ileczekas−−;
   }

4. **if** (ileczekas > 0) **V**(Stos)
   else **V**(Monitor);

Procesy wykonujące *wait*() i kończące procedurę monitora budzą i przekazują sekcję krytyczną procesom, które wykonały *signal*() oczekującym na semaforze *Stos*. Nie jest to oczywiście dokładna implementacja, ponieważ za pomocą semaforów nie można symulować stosu. Procesy oczekujące po wykonaniu *signal*() wykonają się przed procesami rozpoczynającymi procedurę monitora, ale nie w wymaganej kolejności.

Zadanie 3: Zasoby

W systemie są dwie grupy procesów korzystające z N zasobów typu A i M zasobów typu B ($N + M > 1$). Procesy z pierwszej grupy cyklicznie wykonują własne sprawy, po czym wywołują procedurę *zamieńAB*(), która konsumuje jeden zasób A i produkuje jeden zasób B. Procesy z grupy drugiej cyklicznie wykonują własne sprawy, po czym wywołują procedurę *zamień*(), która konsumuje jeden zasób dowolnego typu i produkuje zasób przeciwny. Zsynchronizuj procesy za pomocą monitora tak, aby:

- procedury *zamieńAB*() i *zamień*() były wywoływane przez procesy jedynie pod warunkiem dostępności odpowiednich zasobów,
- procesy z grupy pierwszej wykonywały procedurę *zamieńAB*() parami, tzn. proces z grupy pierwszej może rozpocząć wykonanie procedury jedynie wtedy, gdy jest inny proces z grupy pierwszej gotowy do jej wykonania (i oczywiście niezbędne zasoby),
- jednocześnie mogło odbywać się wiele operacji na zasobach, ale nie doszło do zagłodzenia żadnej grupy procesów.

Monitor powinien udostępniać jedynie procedury wywoływane przez procesy przed i po rozpoczęciu korzystania z zasobów.

Rozwiązanie

Przyjmijmy, że procesy z grupy pierwszej przed zamianą wykonują procedurę *ChceA*() monitora, a procesy grupy drugiej – *ChceZasób*(), która przekazuje w wyniku typ przyznanego zasobu. Po wykonaniu zamiany wszystkie procesy wykonują *Wyprodukowałem*(*Zasób z*) podając typ zasobu, który został wyprodukowany.

```
monitor Zasoby{

  int ileA  = N;
  int ileB  = M
  int iluNaA = 0;
  condition zasóbA, zasób, partner;

  void ChceA() {
    iluNaA++;
    if (iluNaA % 2 == 1) wait(partner) /* pierwszy z pary czeka na partnera */
    else {                             /* drugi z pary */
      if (ileA < 2)                    /* jeżeli nie ma dwóch zasobów A */
        wait(zasóbA);                  /* czeka na co najmniej dwa zasoby A */
      ileA = ileA - 2;
      signal(partner);                 /* zwalnia partnera */
    }
    iluNaA--;
  }

  Zasób ChceZasób() {
    if ((ileB == 0) && (ileA == 0 || iluNaA >= 2))
      wait (zasób);                    /* czeka na dowolny zasób */
    if (ileB > 0) {                    /* najpierw B, żeby produkować A */
      ileB--;
      return(B);
    }
    else {
      ileA--;
      return(A);
    }
  }

  void Wyprodukowałem(Zasób z) {
    if (z == B) {
      ileB++;
      signal(Zasób);
    }
    else {
      ileA++;
      if (empty(ZasóbA))     /* tylko jeżeli nie ma par w pierwszej grupie */
        signal(Zasób);       /* to zwalniamy drugą grupę */
      else                   /* czeka para na zasób A */
        if (ileA > 1)        /* zwalniamy ją, jeśli można */
          signal(ZasóbA);    /* wpp czekamy na drugi egzemplarz zasobu A */
    }
  }
}
```

Aby nie zagłodzić procesów pierwszej grupy, proces który wyprodukował A, zwalnia procesy drugiej grupy tylko wtedy, gdy nie ma pary gotowych procesów z grupy pierwszej, nawet jeżeli jest tylko jeden zasób, który na razie jest dla tej pary bezużyteczny.

Alternatywne rozwiązanie

```
monitor Zasoby{

  int ileA = N;
  int ileB = M
  condition zasóbA, zasób, partner;

  void ChceA() {
    if (ileA == 0) wait(zasóbA);          /* zapewniam sobie zasób */
    ileA--;
    if (empty(partner)) wait(partner);    /* pierwszy z pary czeka na partnera */
    else signal(partner);                 /* drugi zwalnia partnera */
  }

  Zasób ChceZasób() {
    if (ileB + ileA == 0)
      wait (zasób);                       /* czeka na dowolny zasób */
    if (ileB > 0) {                       /* najpierw B, żeby produkować A */
      ileB--;
      return(B);
    }
    else {
      ileA--;
      return(A);
    }
  }

  void Wyprodukowałem(Zasób z) {
    if (z == B) {
      ileB++;
      signal(Zasób);
    }
    else {
      ileA++;
      if (empty(ZasóbA))    /* tylko jeżeli nie ma oczekujących z pierwszej grupy */
        signal(Zasób);      /* zwalniamy drugą grupę */
      else
        signal(ZasóbA);
    }
  }
}
```

W drugim rozwiązaniu proces z pierwszej grupy rezerwuje dla siebie zasób, a dopiero potem czeka na partnera.