Synchronizacja procesów

Wstęp

Współbieżny dostęp do danych może powodować ich niespójność.

Aby temu zapobiec konieczne są odpowiednie mechanizmy zachowania spójności przez uporządkowane wykonanie procesów współpracujących.

Próba rozwiązania problemu producenta/konsumenta z ograniczonym buforem w pamięci wspólnej.

```
#define BUFFER_SIZE 10
Typedef struct {
 ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Schemat procesu producenta:

```
item nextProduced;

while(1){
     /* produkuj kolejny element */
     while( counter == BUFFER_SIZE )
                ; /* nic nie rób */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Schemat procesu konsumenta:

```
item nextConsumed;
while(1){
      while( counter == 0 )
            ; /* nic nie rób */
      nextConsumed = buffer[out];
      out = (out+1) % BUFFER_SIZE;
      counter--;
      /* konsumuj kolejny element */
}
```

Poprawność zależy od poprawnej wartości zmiennej `counter`.

Instrukcje zmiany wartości licznika na poziomie maszynowym są realizowane np. tak:

```
register1 = counter
register1 = register1 + 1
counter = register1

register2 = counter
register2 = register2 - 1
counter = register2
```

W przypadku wykonania współbieżnego procesów konsumenta i producenta możliwy jest przeplot tych instrukcji.

Przeplot ten zależy od sposobu planowania (szeregowania) w systemie operacyjnym i w praktyce może być dowolny.

Niech wartość początkowa licznika będzie 5.

Jedna z możliwości przeplotu to:

```
producer: register1 = counter (register1 = 5)
```

```
producer: register1 = register1 + 1 (register1 = 6)
consumer: register2 = counter (register2 = 5)
consumer: register2 = register2 - 1 (register2 = 4)
producer: counter = register1 (counter = 6)
consumer: counter = register2 (counter = 4)
```

Wartość licznika może być albo 4 albo 6 chociaż poprawna powinna być 5.

Aby zapobiec takim efektom operacje `counter++` i `counter--` muszą być atomowe (niepodzielne).

Szkodliwa rywalizacja — sytuacja kiedy kilka procesów wykonuje operacje na tych samych danych i wynik ostateczny zależy od porządku w jakim operacje zostały wykonane.

Aby temu zapobiec konieczna jest synchronizacja, np. zagwarantowanie, że tylko jeden proces naraz ma dostęp do zmiennej `counter`.

Problem sekcji krytycznej

Mamy `n` procesów korzystających ze wspólnych danych.

Każdy z procesów ma segment kodu zwany *sekcją krytyczną*, w którym realizuje dostęp do wspólnych danych.

Problem — zapewnienie, że kiedy jeden z procesów realizuje swoją sekcję krytyczną, żaden inny proces nie realizuje swojej.

Rozwiązanie problemu musi spełniać warunki:

1. Wzajemne wykluczanie — jeśli proces P_i działa w swojej sekcji krytycznej, to żaden inny proces nie działa w sekcji krytycznej.
2. Postęp — jeśli żaden proces nie działa w sekcji krytycznej i są chętni do wejścia do sekcji krytycznej to wybór procesu, który wejdzie do sekcji nie może być odwlekany w nieskończoność.

3. Ograniczone czekanie — musi istnieć ograniczenie liczby wejść innych procesów do sekcji krytycznej po tym, gdy dany proces zgłosił chęć wejścia do sekcji i zanim uzyskał na to pozwolenie.

Zakładamy, że każdy proces jest wykonywany z niezerową prędkością.

Nic nie zakładamy o względnej prędkości wykonywania procesów.

Ogólna struktura procesu może być taka:

```
do {
     entry section
         critical section
     exit section
         reminder section
} while (1);
```

Naszym celem jest zaimplementowanie sekcji wejściowej i wyjściowej tak, aby spełnione zostały wszystkie warunki.

Procesy mogą współdzielić pewne zmienne aby zsynchronizować swoje działanie.

Rozwiązania dla 2 procesów (P_0 i P_1)

Algorytm 1

Zmienna dzielona:

`int turn;`

Początkowo:

`turn = 0`

Jeśli `turn == i` to P_i może wejść do sekcji krytycznej.

Process P_i:

```
do {
   while (turn != i);
      critical section
   turn = j;
      reminder section
} while (1);
```

Rozwiązanie spełnia warunek wzajemnego wykluczania ale nie spełnia warunku postępu.

Algorytm 2

Zmienne dzielone:

```
boolean flag[2];
```

Początkowo:

`flag [0] = flag[1] = false`.

Jeśli `flag[i] = true` to proces P_i jest gotów do wejścia do sekcji krytycznej.

Process P_i:

```
do{
       flag[i] = true;
       while ( flag[j] );
          critical section
       flag[i] = false;
          remainder section
  } while (1);
```

To rozwiązanie również spełnia warunek wzajemnego wykluczania ale nie spełnia warunku postępu.

Algorytm 3

Algorytm łączy zmienne wspólne wersji 1 (komu wolno wejść) i wersji 2 (kto chce wejść). Dopiero połączenie chęci z zezwoleniem daje rozwiązanie poprawne.

Process P_i:

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] and turn == j);
            critical section
        flag[i] = false;
            remainder section
} while (1);
```

To rozwiązanie (Dekker) spełnia wszystkie trzy warunki.

Algorytmy dla wielu procesów

Możliwe jest uogólnienie algorytmu pokazanego wyżej.

Inna możliwość to tak zwany algorytm piekarniany (u nas stoswany na poczcie).

Polega na przydzielaniu procesom numerków. Obsługiwany jest proces z najmniejszym numerkiem. Schemat numerowania gwarantuje niemalejący porządek numerków. W razie kolizji bierze się pod uwagę identyfikator procesu.

Przyjmijmy notację:

$(a, b) < (c, d)$ wtw jeśli $a < c$ lub $a = c$ i $b < d$

$max(a_0, ..., a_n)$ — maksymalny element spośród $a_1, ..., a_n$

Dzielone zmienne:

```
boolean choosing[n]; int number[n];
```

inicjalizowane na `false` i `0` odpowiednio.

```
do {
  choosing[i] = true;
  number[i] =
      max(number[0], number[1], ..., number [n-1])+1;
  choosing[i] = false;
  for (j = 0; j < n; j++) {
       while (choosing[j]) ;
       while ((number[j] != 0)
           && ((number[j],j) < (number[i],i)) ;
  }
    critical section
  number[i] = 0;
    remainder section
  } while (1);
```

Synchronizacja sprzętowa

Istnienie odpowiednich mechanizmów sprzętowych ułatwia rozwiązanie problemu sekcji krytycznej.

Blokowanie przerwań nie jest przydatne w środowisku wieloprocesorowym.

Specjalne rozkazy sprzętowe — w sposób niepodzielny sprawdzają i

zmieniają zawartość słowa albo zamieniają zawartość dwóch słów.

```
boolean TestAndSet(boolean &target)
{ boolean rv = target;
  target = true;
  return rv;
 }
```

Wzajemne wykluczanie

Dzielona zmienna: `boolean lock = false;`

Process P_i

```
do {
   while (TestAndSet(lock)) ;
     critical section
   lock = false;
     remainder section
 }
```

Druga instrukcja tego typu to:

```
void Swap(boolean &a, boolean &b)
 { boolean temp = a;
   a = b;
   b = temp;
 }
```

Wzajemne wykluczanie ze swap

Zmienna dzielona `boolean lock;` inicjalizowana na `false`.

Process P_i

```
boolean key; /* lokalna w procesie */
do
 { key = true;
   while (key == true) Swap(lock,key);
     critical section
   lock = false;
     remainder section
}
```

Oba powyższe algorytmy mają dwie wady:

1. Nie spełniają warunku ograniczonego czekania — da się to poprawić.
2. Wykorzystują aktywne czekanie.

Semafory

Semafor — zmienna o wartościach całkowitych dostępna wyłącznie przez dwie niepodzielne (atomowe) operacje (plus ewentualnie inicjalizacja):

```
  wait(S): while S <=  0 do noop;
           S--;
signal(S): S++;
```

Tradycyjnie operacje te oznaczane są jako P(S) i V(S).

Sekcja krytyczna n procesów

Zmienna globalna `semaphore mutex; //initially mutex = 1`

Process P_i:

```
do {
   wait(mutex);
      critical section
   signal(mutex);
      remainder section
  } while (1);
```

Implementacja semafora

Klasyczna definicja zawiera pojęcie aktywnego czekania — zwykle uważane za niekorzystne.

Spinlock (wirująca blokada) — ich zaletą jest eliminacja przełączania kontekstu, które może trwać długo.

Uniknięcie aktywnego czekania umożliwia zmiana definicji operacji semaforowych.

Zamiast aktywnego czekania proces wykonuje operację blokowania — umieszczenie procesu w kolejce związanej z semaforem i zmianę jego stanu na czekanie (uśpienie).

Wznowienie procesu następuje w wyniku wykonania operacji budzenia przez inny proces.

```
typedef struct {
                  int value;
                  struct process *L;
                } semaphore;
```

Operacje semaforowe definiujemy:

```
  wait(S): S.value--;
           if (S.value < 0) {
```

```
            add this process to S.L;
            block;
           }
signal(S): S.value++;
          if (S.value <= 0) {
            remove a process P from S.L;
            wakeup(P);
           }
```

Tylko niepodzielne wykonanie gwarantuje poprawne działanie semaforów.

Blokada i zagłodzenie

Blokada (zakleszczenie) — zbiór procesów, z których każdy czeka na zdarzenie, które może spowodować tylko inny proces z tego zbioru.

Niech w systemie będą dwa procesy:

```
  P 0          P 1
wait(S);    wait(Q);
wait(Q);    wait(S);
...         ...
signal(S);  signal(Q);
signal(Q);  signal(S);
```

Zagłodzenie — nieskończone oczekiwanie pod semaforem — np. niewłaściwa strategia obsługi kolejek związanych z semaforami.

Niektóre klasyczne problemy synchronizacyjne

- ograniczone buforowanie,

- problem czytelników i pisarzy,
- problem pięciu filozofów.

Producent i konsument z ograniczonym buforem

Zmienne globalne:

```
semaphore full, empty, mutex;
```

Zainicjowane:

```
full = 0, empty = n, mutex = 1
```

Proces producenta:

```
do
  { ...
    produce an item in nextp
    ...
    wait(empty);
    wait(mutex); ...
    add nextp to buffer
    ...
    signal(mutex);
    signal(full);
 } while (1);
```

Proces konsumenta jest symetryczny:

```
do
   { wait(full)
     wait(mutex);
     ...
     remove an item from buffer to nextc
```

```
    ...
    signal(mutex);
    signal(empty);
    ...
    consume the item in nextc
    ...
  } while (1);
```

Problem czytelników i pisarzy

Obiekt danych (np. plik) jest współdzielony przez procesy. Niektóre z nich tylko czytają plik (czytelnicy), natomiast inne mogą go modyfikować (pisarze).

Czytelnicy mogą czytać plik równocześnie z innymi czytelnikami.

Pisarze muszą mieć dostęp wyłączny do pliku.

Różne warianty — priorytet czytelników, priorytet pisarzy, brak uprzywilejowania.

Zmienne globalne:

```
semaphore mutex, wrt;
int readcount;
```

Zainicjowane:

```
mutex = 1, wrt = 1, readcount = 0
```

Proces pisarza:

```
wait(wrt);
  ...
  writing is performed
  ...
```

```
signal(wrt);
```

Proces czytelnika:

```
wait(mutex);
  readcount++;
  if (readcount == 1) wait(rt);
signal(mutex);
  ...
  reading is performed
  ...
wait(mutex);
  readcount--;
  if (readcount == 0) signal(wrt);
signal(mutex):
```

Który wariant problemu realizuje algorytm?

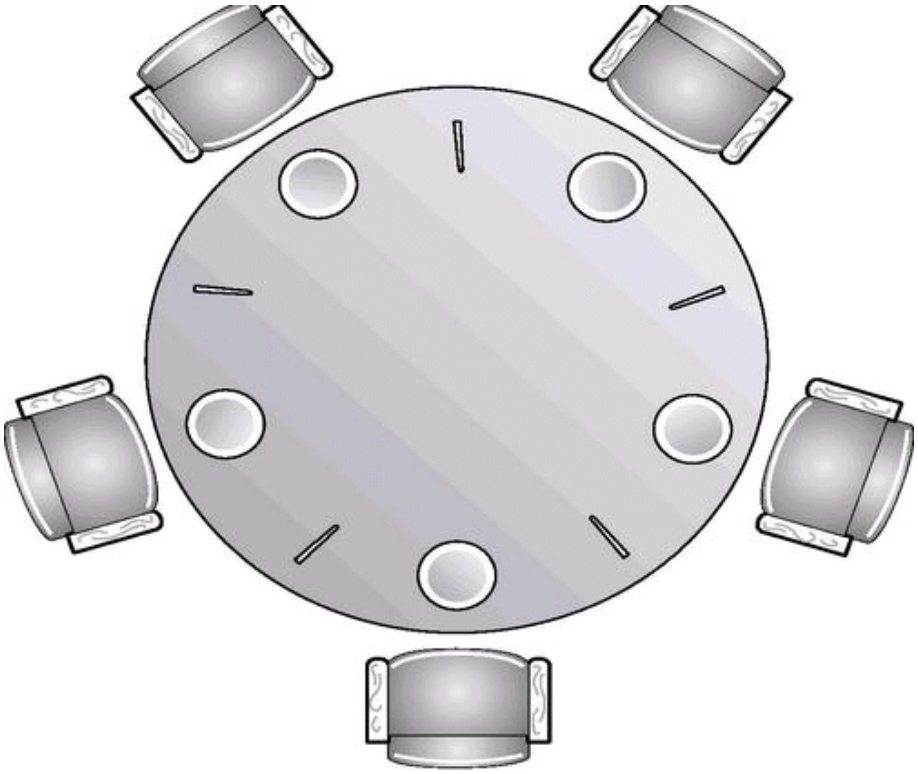
Problem pięciu filozofów — wersja orientalna

Filozofowie zajmują się myśleniem. Od czasu do czasu odczuwają głód i wtedy siadają przy stole, ujmują najpierw jedną pałeczkę, potem drugą, posilają się, odkładają pałeczki i wstają.

Przyjmijmy, że każda pałeczka jest semaforem:

```
semaphore chopstick[5];
```

zainicjowanym na wartość 1.



Proces filozofa może mieć postać:

```
do {
  wait(chopstick[i])
  wait(chopstick[(i+1) % 5])
    ...
    eat
    ...
  signal(chopstick[i]);
  signal(chopstick[(i+1) % 5]);
    ...
    think
    ...
  } while (1);
```

To rozwiązanie jest nieakceptowalne ze względu na możliwość blokady.

Różne sposoby uniknięcia problemu:

- Siada naraz nie więcej niż 4 filozofów.
- Pałeczki podnoszone są tylko po dwie jednocześnie.
- Rozwiązanie asymetryczne — niektórzy zaczynają prawą a niektórzy lewą ręką.